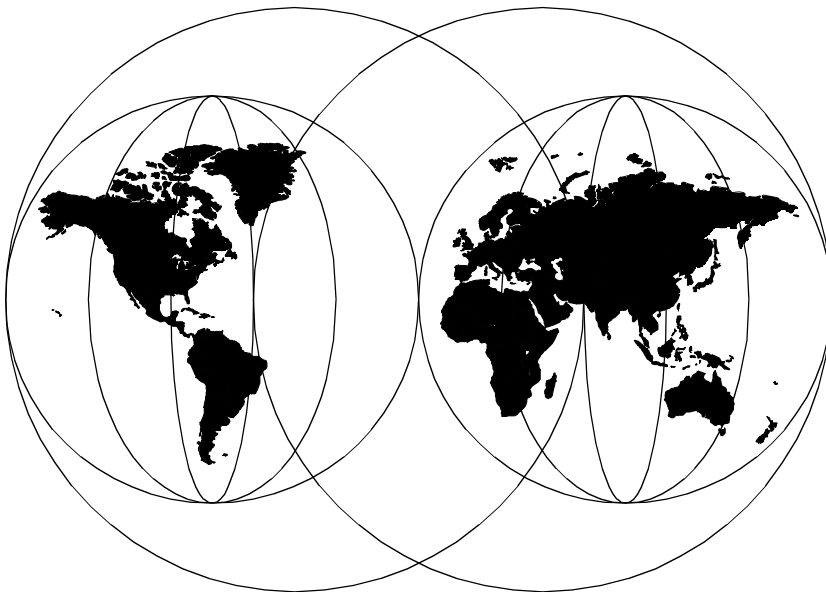


Application Development with VisualAge for Java Enterprise

*Ueli Wahli Stefania Celentano Werner Frei
Bruno Georges Paul Gover Rudolf Wirawan*



International Technical Support Organization

<http://www.redbooks.ibm.com>



International Technical Support Organization

Application Development with VisualAge for Java Enterprise

April 1998

Take Note!

Before using this information and the product it supports, be sure to read the general information in Appendix C, “Special Notices” on page 383.

First Edition (April 1998)

This edition applies to Version 1.0 of VisualAge for Java Enterprise, Program Number 5801-AAR, for use with the OS/2, Windows 95, or Windows NT operating system.

SAMPLE CODE ON THE INTERNET

The sample code for this redbook is available as sg245081.zip on the ITSO home page on the Internet:

<ftp://www.redbooks.ibm.com/redbooks/SG245081>

Download the sample code and read “Installation of the Redbook Samples” on page 368.

Comments may be addressed to:
IBM Corporation, International Technical Support Organization
Dept. QXXE Building 80-E2
650 Harry Road
San Jose, California 95120-6099

When you send information to IBM, you grant IBM a non-exclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

© Copyright International Business Machines Corporation 1998. All rights reserved

Note to U.S Government Users – Documentation related to restricted rights – Use, duplication or disclosure is subject to restrictions set forth in GSA ADP Schedule Contract with IBM Corp.

Contents

Figures	xv
Tables	xxi
Preface	xxiii
How This Document Is Organized	xxiv
The Team That Wrote This Redbook	xxvi
Acknowledgments	xxvii
Comments Welcome	xxviii
 Chapter 1. Introducing VisualAge for Java Enterprise	1
VisualAge for Java Products	2
VisualAge for Java Professional	2
VisualAge for Java Entry	2
VisualAge for Java Enterprise	3
Team Programming Feature	3
Enterprise Access Builders	4
Data Access Builder Overview	6
Why a Data Access Builder?	7
RMI Access Builder Overview	8
Why an RMI Access Builder?	9
CICS Access Builder Overview	10
C++ Access Builder Overview	12
VisualAge for Java Enterprise Connectivity	13
Applet Connectivity	13
Application Connectivity	14
Connecting to Other Servers	15
Connecting to VisualAge Generator	15
Connecting to Component Broker	16
 Chapter 2. Sample ATM Application and Database	17
ATM Application Requirements	18
Application Design	20
User Interface	20
Data Store	20
Business Logic	20
Database Design	21
Logical Design	21
Reducing Redundancy and Eliminating Anomalies	23
Referential Integrity	24
Applying Referential Integrity to the ATM Application	27

Physical Design	27
ATM Database	29
Sample Data of ATM Tables	30
Chapter 3. Java Database Connectivity	33
Database Application Architectures	34
The Ideal Solution for Programmers	34
Single-Tier Architecture	35
Two-Tier Architecture	36
Three-Tier Architecture	37
JDBC	38
JDBC Drivers	39
The Structure of an Ideal JDBC Driver	39
Building on Existing Products	41
JDBC and ODBC Bridge Driver	41
JDBC and Vendor-Specific Bridge Driver	42
JDBC Generic Network Protocol Driver	43
Structure of a JDBC Application	44
JDBC Connection Sample	45
JDBC Applications and JDBC Applets	47
JDBC Application	48
JDBC Applet	50
JDBC Sample for Insert, Update, and Delete	51
Statement and Prepared Statement	54
Callable Statement	55
JDBC in VisualAge for Java Enterprise	57
Chapter 4. Data Access Builder	59
Relational Database Access	60
Building a JDBC Application	61
Application Requirements	61
Development Process with Data Access Builder	63
Using Data Access Builder for the Organization Applet	64
Starting Data Access Builder	65
Mapping a Table into Data Access Beans	66
Generating the Data Access Beans	70
Data Access Builder Beans and Classes	72
Creating the JDBC Sample Application and Applet	79
Applet Overview	80
Applet Construction	81
Connecting to the Database	84
Completing the Organization List Panel	85
Completing the Organization Detail Panel	87
Data Access Beans in Handwritten Programs	89
Data Access Builder Advanced	92

Sharing Mappings among Developers	92
Running Data Access Builder Stand-Alone	92
Interesting Methods of the Manager Bean	94
Eliminating Attributes from the Mapping	94
Customized SQL Statements	94
Encapsulating an SQL Search Predicate	95
Asynchronous Processing	96
Working with Stored Procedures	96
Application Design Considerations	97

Chapter 5. ATM Application with Data Access Builder and JDBC 99

Designing the ATM Application	100
Building the ATM Application	101
Database Classes	101
PIN Validation	101
List of Accounts	105
Account Information	105
Transaction History	105
Adding User-Defined Methods	106
Generating the Data Access Beans	108
Business Logic Classes	109
Card Class	110
BankAccount Class	112
CheckingAccount Class	114
SavingAccount Class	114
User Interface Classes	115
CardPanel Class	115
PinPanel Class	119
SelectAccountPanel Class	121
TransactionPanel Class	125
Application Flow	129
Applet Layout	130
Panel Switching	131
Sharing the Card Object	135
Database Connection	136
Running the ATM Application	136

Chapter 6. Remote Method Invocation and RMI Access Builder . 137

Overview	138
Using RMI for Distributed Processing	139
How Does RMI Work?	140
Squeezing Objects through a Network	140
RMI Architecture	141
Tools	143
RMI Compiler	143

RMI Registry	143
RMI Development Process	143
Special Coding	145
Execution Environment	146
Native RMI Example	147
Public Interface of the Server	147
Server Implementation	147
Stub and Skeleton	149
Client Logic	149
Run the RMI Application	151
Stop the RMI Application	151
More on Native RMI	151
RMI with VisualAge for Java	152
RMI Access Builder	152
Development Process	152
Created Classes and Interfaces	154
RMI Execution Environment with VisualAge for Java	155
Using the RMI Access Builder	157
Create the Server	157
Create an Applet	157
Generate Proxy Bean	158
Connect the Client with the Server	161
Run the RMI Applet	163
Stop the Server	165
RMI Problems and Hints	165
Running an RMI Application outside VisualAge for Java	166
Export of Application Code	166
Start the Registry and the Server	166
Run the Applet	167
Before You Use RMI to Build a Distributed Application	167
Design Considerations	167
Limitations	169
Chapter 7. ATM Application with RMI	171
Design for Distribution	172
Application Layers	172
Application Layer Architecture	173
Business Object Layer	174
BankAccount	175
CheckingAccount	176
SavingsAccount	176
Card	177
Customer	177
Transaction	178
Testing the Business Objects	178

System Service Layer	179
Creating the Data Access Beans	179
Tailoring the Data Access Beans	180
Generating the Beans	182
Initialize Business Objects from Data Access Beans	182
Creating Transaction Data Access Beans from Business Objects	183
Connecting the Layers	183
Data Access Classes for Business Objects	184
Customer Access Class	185
Card Access Class	186
Account Access Class	187
Transaction Access Class	190
ATM Service Bean	192
Controller	195
Controller Features	195
Controller Methods and Events	197
Event Propagation	199
External Interface	201
Testing the Beans	201
View Layer	202
CardPanel Class	204
PinPanel Class	205
SelectAccountPanel Class	206
TransactionPanel Class	208
Main Panel	210
Testing the Stand-Alone Applet	211
Distributed ATM Application	213
Application Changes	213
Make the Beans Serializable	213
Mark the Methods That Update the Bank Account As Synchronized	213
Review the Events	214
Create the Proxy Beans	215
Modify the GUI	216
Using the Distributed Controller	216
Changes in Subpanels	217
Test the Distributed ATM Application	219
Running the Applet on a Client	219
Running As an Application	220
Chapter 8. Host CICS Access with the CICS Access Builder	221
Host CICS Access Overview	222
CICS	222
CICS Gateway for Java and CICS Access Builder	222
How Does the CICS Access Builder Work?	223
Working with a CICS Enterprise Server	224

CICS Java Application Design	225
CICS Access Builder	226
CICS Access Builder: Overview	226
Create COMMAREA Bean SmartGuide	226
Run-Time Class Library	227
ATM Application with the CICS Access Builder	229
CICS Program Design	230
Accessing the Database	231
Building the CICS Programs	232
COBOL Input to the CICS Access Builder	232
Restrictions	234
Data Types and Non-COBOL Programs	234
Running the CICS Access Builder	237
Running the CICS Access Builder from the Workbench	237
Running the CICS Access Builder from the Command Line	239
Generated Classes	241
Application Coding Techniques	241
Throwing Exceptions from the CICS Host	241
Exchanging Array Data with CICS	242
Sample COBOL CICS Transaction	244
ATM Applet Using Host CICS Access	246
Visual Composition	246
Properties of the CICS Unit of Work Bean	248
Simulating a CICS Server	249
Installing CICS and Java Components	250
Current Restrictions	251
CICS Host Access Topologies	252
CICS Gateway for Java and CICS Client Topologies	252
Client/Server Tier Topology	252
Presentation Logic Tier	253
Data Logic Tier	253
Business Logic Tier	253
Chapter 9. C++ Servers and C++ Access Builder	255
Java Native Interface Overview	256
When to Use?	256
Java Native Interface Programming	257
Declaring and Loading Native Methods	257
Simple JNI Example	258
JNI Development Process	261
Type Mapping between Java and C/C++	262
Accessing Java Methods and Fields from Native Code	262
Exception Handling	263
Object References and Java Garbage Collector	263
How to Make Your Life Easier?	264

C++ Access Builder Overview	265
High-level View	265
Command Line Utility	266
Revisiting the Native Example with the C++ Access Builder	267
Reverse String Example	269
C++ Access Builder Advanced	272
Considerations for C++ Class Wrappering	272
Details of the Generated Code	273
Design Considerations	276
Type Mapping between C++ and Java	277
Exception Handling	279
Compiler Support	280
Limitations of the C++ to Java Mapping	280
Another Way of Exposing the C++ Interfaces	281
Accessing a Complex Class by Header File Modification	281
Using a Class That Accesses a Wrapped C++ Library	284
Accessing a Class with Templates	287
C++ Access Builder Supported Environments	289
Using a C++ Server in the ATM Application	290
Environment	290
Approach	291
C++ Header Files	291
Mapping the ATM C++ Classes to Java	292
Mapping the CardManager Class	293
Mapping the Card Class	294
Testing the Card Beans	297
Wrapping the Beans for the ATM Application	299
C++ Card Server Bean	299
Testing the C++ Card Server Bean	301
Integrating the C++ Card Server into the ATM Application	301
 Chapter 10. Access to VisualAge Generator Servers	303
VisualAge Generator Support for Java	304
VisualAge Generator	304
VisualAge Generator Java Support	306
Implementation of Java Support	307
VisualAge Generator CSO Java Classes	308
VisualAge Generator Generated Java Beans	308
Accessing VisualAge Generator Servers from Java Clients	308
From a Java Application	309
From a Java Applet	310
The e-Business Solution: Java and VisualAge Generator	313
Run-time Configuration for Java Applets and VisualAge Generator Servers ..	313
Value of VisualAge Generator in an e-Business Solution	314
ATM Application with a VisualAge Generator Server	315

Chapter 11. Access to Distributed CORBA Objects	317
The Case for CORBA	318
Why CORBA?	318
What Is CORBA?	321
Object Management Group	321
Object Request Broker	322
Interface Definition Language	323
IIOP Communication Protocol	325
CORBA Services	325
CORBA and RMI	329
How Java Complements CORBA	329
Component Broker	330
CBConnector	330
CB Toolkit	330
Run-time Architecture Components	331
Programming Model	331
Managed Object Framework	331
Application Adapter Framework	333
Object Services	333
Workload Management	334
Client Enablement	334
Developing Distributed Object Applications with Component Broker	335
Modeling, Analysis, and Design	335
Object Builder	335
Edit, Compile, and Debug	336
Systems Management	336
Java Client Accessing a CBConnector Server	337
Account Interface Definition	337
Account Development with CBConnector	338
Java Client	338
Preparing VisualAge for Java	338
Generating Java Proxies	338
Creating the Java Client	341
Creating the Java Bean for the Applet	342
Creating the Applet	346
Creating Account Objects	347
Updating Account Objects	349
Finding Account Objects	350
Releasing and Deleting Objects	351
 Chapter 12. Deployment of Java Applications and Applets	 353
Deployment of Applications	354
Prerequisites for Applications	354
Design for Portability	354
Exporting an Application from VisualAge for Java	354

Deployment Process for Applications	355
Deployment of Applets	356
Exporting an Applet from VisualAge for Java	356
Deployment Process for Applets	356
Run-time Libraries	358
Jar Files	358
Appendix A. Installation, Setup, and Prerequisites	359
Prerequisites for JDBC Applications	360
DB2 Prerequisites	360
VisualAge for Java Prerequisites	360
ODBC Prerequisites	361
Installation and Setup of CICS Components	362
Installing the CICS Client for Windows NT	362
Configuring the CICS Client for TCP/IP Connections	363
Configuring the CICS Client for APPC Connections	364
Installing the CICS Gateway for Java	367
Installation of the Redbook Samples	368
Appendix B. Enterprise Access Builder Changes in Version 1.0.1	371
AS/400 Feature	372
Data Access Builder	373
CICS Access Builder	374
Changes to the IVJCicsUOWInterface Class	374
Additional Version of Transaction Invocation	374
DFHCNV Support	374
Closing the CICS Gateway for Java	375
Setting the CICS Transaction ID	375
Specifying a Program to Execute	376
Changes to IVJCicsEciCommArea Bean	377
Setting the CICS Transaction ID	377
Double-Byte Character Set Support	378
Changes to Limitation on Code Page	378
C++ Access Builder	379
Compatibility between Versions	379
Deleting C++ Objects Allocated in Java	379
Character Arrays	379
Signed Characters	380
Pointers	380
Exceptions	380
Migration to Version 1.0.1?	381

Appendix C. Special Notices	383
Appendix D. Related Publications	387
International Technical Support Organization Publications	388
Redbooks on CD-ROMs	389
Other Publications	390
How To Get ITSO Redbooks	391
How IBM Employees Can Get ITSO Redbooks	391
How Customers Can Get ITSO Redbooks	392
IBM Redbook Order Form	393
Glossary	395
List of Abbreviations	405
Index	407
ITSO Redbook Evaluation	415

Figures

1. Development Process Using an Enterprise Access Builder	4
2. Enterprise Access Builder Overview	5
3. JDBC Database Access Configurations	6
4. Data Access Builder Overview	7
5. RMI Overview	8
6. RMI Access Builder Overview	9
7. CICS Gateway for Java	10
8. CICS Access Builder Overview	11
9. C++ Access Builder Overview	12
10. Applet Connectivity to Enterprise Servers	13
11. Application Connectivity to Enterprise Servers	14
12. Java Applet with VisualAge Generator Server	16
13. ATM Application Panels and Flow	19
14. ATM Entity-Relationship Diagram	22
15. Table with Third Normal Form Violation	24
16. Table in Third Normal Form	24
17. Referential Integrity Constraints in the ATM Database	25
18. ATM Database Data Definition Language	29
19. A Banking Application with Two Different Databases	34
20. Data Access Layers: Generic and DBMS-Specific	34
21. Open Database Connectivity Architecture	35
22. Monolithic (Single-Tier) Architecture	35
23. Client/Server (Two-Tier) Architecture	36
24. Multiple Client Instances Accessing One Server	36
25. Dividing the Application into Client and Server	37
26. Three-Tier Architecture	37
27. Data Access Layer Revised	38
28. JDBC Applet in the Intranet and Internet Environment	39
29. An Ideal JDBC Driver	40
30. JDBC-ODBC Bridge Driver	41
31. Vendor-Specific Driver	42
32. JDBC Vendor Specific Bridge	42
33. JDBC Generic Network Protocol Driver	43
34. Structure of JDBC Application	44
35. Sample JDBC Program	46
36. DB2 Sample Organization Table	47
37. JDBC DB2 Organization Application	48
38. JDBC DB2 Organization Applet	50
39. JDBC Applet in the AppletViewer	51
40. JDBC Insert, Update, and Delete Program	52
41. Statement and PreparedStatement Classes	54
42. Simple JDBC Applet GUI Design	61
43. The JDBC Applet Result	62
44. Application with Ready-to-Use GUI Components	62
45. Development Process with Data Access Builder	63
46. JDBC Project in the Workbench	64

47. Launching Data Access Builder	65
48. Data Access Builder	66
49. Database and Mapping Method Selection	66
50. Table Mapping Selection	67
51. Generated Org Bean and Attributes	68
52. Specifying a Unique Data Identifier	69
53. Org Bean with Unique Data Identifier	69
54. Connection Information in the Org Bean Properties Window	70
55. Beans and Classes Generated by Data Access Builder	71
56. Forms Generated by Data Access Builder	74
57. Database Connection Panel	75
58. Access Application: Datastore Page	76
59. Access Application: Org (Row Manipulation)	77
60. Access Application: Manager	77
61. Access Application: Cursor	78
62. Access Application: ResultForm	78
63. Access Application: Selected	79
64. Organization Applet: Main Panel	80
65. Organization Applet: Selecting a Row	80
66. Organization Applet: Selected Row	81
67. Organization Applet: User Interface Design	82
68. Organization Applet: Organization List Panel (PanelA)	83
69. Organization Applet: Organization Detail Panel (PanelB)	83
70. Organization Applet: CardLayout Connectivity	84
71. Organization Applet: Database Connection	85
72. Organization Applet: Organization List Connections	87
73. Organization Applet: Organization Detail Connections	88
74. Handwritten Program Using Data Access Beans	90
75. Data Access Builder: Startup Window	93
76. Data Access Builder: Save Session	93
77. Object Model of the ATM Application	100
78. Data Access Builder SmartGuide Dialog	102
79. Selecting the Tables for Mapping	103
80. Validating the SQL Statement	104
81. Database Access Builder Window: Mappings	106
82. Context Menu of a Dax Bean	107
83. Validating the SQL Statement and Setting Parameter Names	108
84. Defining an Event with an Event Listener	111
85. Layout of the CardPanel	116
86. Visual Composition Editor: CardPanel	117
87. Layout of the PinPanel	119
88. Visual Composition Editor: PinPanel	120
89. Layout of the SelectAccountPanel	121
90. Visual Composition Editor: SelectAccountPanel	122
91. Layout of the TransactionPanel	125
92. Visual Composition Editor: TransactionPanel	126
93. ATM Applet Main Panel and Beans	130
94. ATM Applet Beans List	131
95. CardPanel Reorder Connections Window	132

96. PinPanel Reorder Connections Window	133
97. SelectAccountPanel Reorder Connections Window	134
98. TransactionPanel Reorder Connections Window	134
99. Card Variable Reorder Connections Window	136
100. Two-Tier Architecture	138
101. Three-Tier Architecture with JDBC Network Protocol Driver	138
102. Three-Tier Architecture with Client/Server Application	139
103. RMI Conceptual View	140
104. RMI Architecture	142
105. RMI Development Process	144
106. RMI Execution Environment	146
107. Development Process with RMI Access Builder	153
108. RMI Server Execution Environment	155
109. RMI Client Execution Environment	156
110. Account Applet before Distribution	158
111. Beans List of Account Applet	158
112. Start Create Proxy Bean SmartGuide	159
113. Create Proxy Bean SmartGuide	160
114. Generated RMI Beans	161
115. AccountApplet after Distribution	162
116. Properties of the RmiAccountBean	162
117. The RMI Options Page	163
118. The Remote Object Instance Manager	164
119. Layers of the ATM Application	173
120. Object Model of the ATM Business Objects	175
121. Data Access Builder Mappings for RMI ATM Application	180
122. Data Access Classes for Business Objects	184
123. Visual Composition of CustomerDB Panel	185
124. Visual Composition of CardDB Panel	186
125. Visual Composition of AccountDB Panel	187
126. Visual Composition of TransactionDB Panel	190
127. The AtmDB Bean	192
128. AtmDB Visual Composition	193
129. Application Controller Interfaces	201
130. ATM Applet Beans List: Old and New	203
131. Visual Composition Editor: CardPanel	204
132. Visual Composition Editor: PinPanel	205
133. PinPanel: Connections from Card	206
134. Visual Composition Editor: SelectAccountPanel	206
135. Visual Composition Editor: TransactionPanel	208
136. Visual Composition Editor: Main Panel	210
137. Sample Run of ATMApplet	212
138. Creating the ATMDistributedController Proxy Bean	215
139. Visual Composition of Distributed ATM Applet	216
140. Visual Composition of the RMI PIN Panel	217
141. Visual Composition of the RMI Transaction Panel	218
142. CICS Access Builder Beans and CICS Gateway for Java	223
143. Three-Tier CICS Java Application	224
144. Sample CICS COMMAREA Structure Definition	232

145. Sample CICS Program Including COMMAREA Definition	233
146. COBOL Example of Most Data Types Supported	237
147. Starting to Create the CICS COMMAREA Bean	238
148. Completing the COMMAREA Bean SmartGuide	238
149. The Result of Creating a COMMAREA Bean	239
150. Running the CICS Access Builder from the Command Line	240
151. Possible Coding for Array Element Get and Set Methods	243
152. COBOL Transaction ATMcard: Declarations	244
153. COBOL Transaction ATMcard: Procedure Division	245
154. Building the CICS Applet	246
155. Populating a List from a COMMAREA Array	247
156. CICS Unit of Work Bean Properties	248
157. Script to Simulate a CICS Transaction	249
158. CICS Server, CICS Client, and CICS Java Gateway	250
159. Simple JNI Example: Java Code	258
160. Simple JNI Example: Generated C Header File	259
161. Simple JNI Example: C Function	260
162. Simple JNI Example: Makefile	260
163. JNI Development Process	261
164. C++ Access Builder Code Generation	266
165. Test Applet for Reverse Bean	271
166. Mapping between C++ and Java Inheritance	273
167. Code for Reverse.java	274
168. Code for ReverseWrapper.cpp	275
169. Reduced IString Header File	282
170. Applet Using the IString Bean	283
171. WordReverse Header and Source Files	285
172. Applet Using the WordReverse Bean	286
173. Full Header File of a SortedList C++ Class	287
174. Reduced SortedList Header File	288
175. CardManager Header File	291
176. Card Header File	292
177. CardManager Interface Class Header File	293
178. CardManager Interface Class Source Code	294
179. Card Interface Class Header File	295
180. Card Interface Class Source File	295
181. Test of Generated Beans for the ATM Application	297
182. Testing the C++ Card Server	301
183. VisualAge Generator Client/Server Support	305
184. Java Client and VisualAge Generator Server Support	307
185. Developing a Java Application	309
186. Developing a Java Applet	311
187. The e-Business Solution: Java Applets and VisualAge Generator Servers ..	313
188. Development and Production Environment Configuration	315
189. Enterprise Access from a Java Client	318
190. The Enterprise Distributed Environment	319
191. Three-Tier Architecture	320
192. The Object Management Architecture	322
193. Client/Server Invocation with IDL	324

194. Component Broker Managed Execution Environment	332
195. AccountView Applet	341
196. Visual Composition of BuildCBC	343
197. Visual Composition of the AccountView Applet	346
198. Creation of Account Objects	347
199. Updating Account Objects	349
200. Finding Account Objects	350
201. Application Deployment Process	355
202. Applet Deployment Process	357

Tables

1. Customer Table.....	28
2. Card Table.....	28
3. Account Table	28
4. Transaction Table.....	28
5. Customer Table Sample Data	30
6. Card Table Sample Data	30
7. Account Table Sample Data.....	30
8. Transaction Table Sample Data	31
9. Beans Generated by Data Access Builder.....	72
10. Form Classes.....	73
11. Bean Feature Table	174
12. BankAccount Features.....	175
13. CheckingAccount Features.....	176
14. SavingAccount Features.....	177
15. Card Features	177
16. Customer Features.....	178
17. Transaction Features	178
18. Controller Features	196
19. Data Types Supported by the CICS Access Builder	235
20. C++ Primitive Type to Java Class Mapping	277
21. VTAM Definitions for the SNA Network in CICS/ESA	365
22. Personal Communications Definitions	365
23. CICS/ESA Definitions.....	366
24. CICS Client INI File Definitions	366
25. VisualAge for Java CICS Unit of Work Bean Properties	367
26. Redbook Sample Code	368
27. Packages of the Redbook Samples	369

Preface

VisualAge for Java Enterprise is the first enterprise-aware, incremental Java application development environment designed to connect Java clients to existing server data, transactions, and applications.

VisualAge for Java Enterprise provides connectivity to enterprise servers on top of the functions of VisualAge for Java Professional. The professional version provides the visual programming paradigm; a workbench with edit, incremental compile, and debug; and an automatic version control system.

The enterprise access builders include a Data Access Builder for JDBC applications that connect to relational databases, an RMI Access Builder that enables construction of distributed Java applications, a CICS Access Builder for applications that invoke CICS transactions, and a C++ Access Builder for applications that invoke C++ server code.

This book demonstrates a practical approach to using VisualAge for Java Enterprise. A sample ATM bank application is used throughout the book to illustrate the use of the enterprise access builders.

How This Document Is Organized

The document is organized as follows:

❑ **Chapter 1, “Introducing VisualAge for Java Enterprise”**

In this chapter we introduce the VisualAge for Java products, VisualAge for Java Enterprise, and the Enterprise Access Builders that enable us to write Java applications that connect to enterprise servers.

❑ **Chapter 2, “Sample ATM Application and Database”**

In this chapter we describe the sample ATM banking application that is used throughout the book to illustrate the function of the VisualAge for Java Enterprise product.

❑ **Chapter 3, “Java Database Connectivity”**

In this chapter we describe the JDBC feature that enables Java applications to access relational databases.

❑ **Chapter 4, “Data Access Builder”**

In this chapter we introduce the Data Access Builder and use it to create JDBC applications.

❑ **Chapter 5, “ATM Application with Data Access Builder and JDBC”**

In this chapter we use the Data Access Builder to implement the ATM application.

❑ **Chapter 6, “Remote Method Invocation and RMI Access Builder”**

In this chapter we describe the remote method invocation (RMI) feature of Java and introduce the RMI Access Builder.

❑ **Chapter 7, “ATM Application with RMI”**

In this chapter we use the RMI Access Builder to implement the ATM application.

❑ **Chapter 8, “Host CICS Access with the CICS Access Builder”**

In this chapter we describe the CICS Gateway for Java and the CICS Access Builder. We implement a part of the ATM application with the CICS Access Builder.

❑ **Chapter 9, “C++ Servers and C++ Access Builder”**

In this chapter we describe the Java native interface to C++ and the C++ Access Builder. We implement a part of the ATM application with the C++ Access Builder.

❑ **Chapter 10, “Access to VisualAge Generator Servers”**

In this chapter we describe how a VisualAge for Java application can use servers generated by VisualAge Generator.

❑ **Chapter 11, “Access to Distributed CORBA Objects”**

In this chapter we describe how a VisualAge for Java application can use CORBA to access a CBBconnector server.

❑ **Chapter 12, “Deployment of Java Applications and Applets”**

In this chapter we describe how applications and applets are deployed from the VisualAge for Java development environment.

❑ **Appendix A, “Installation, Setup, and Prerequisites”**

In this appendix we describe the installation and setup of product components and the sample applications distributed with this document.

❑ **Appendix B, “Enterprise Access Builder Changes in Version 1.0.1”**

In this appendix we describe the enhancement to the Enterprise Access Builders in the national language version of the VisualAge for Java Enterprise product.

❑ **Appendix C, “Special Notices”**

This appendix contains special notices about IBM products and trademarks.

❑ **Appendix D, “Related Publications”**

This appendix contains a list of related publications of the International Technical Support Organization (ITSO) and other sources, and information about how to get ITSO Red-books.

The Team That Wrote This Redbook

Ueli Wahli is a Consultant AD Specialist at the IBM International Technical Support Organization in San Jose, California. Before joining the ITSO 14 years ago, Ueli worked in technical support in IBM Switzerland. He writes extensively and teaches IBM classes worldwide on application development, object technology, VisualAge products, data dictionaries, and library management. He holds a degree in Mathematics from the Swiss Federal Institute of Technology. His e-mail address is *wahli @ us.ibm.com*.

Stefania Celentano is an IT specialist in Italy. She has five years of experience in software development and technical support. She holds a degree in Mathematics from the Federico II University in Naples, Italy. Stefania's areas of expertise include C, C++, object-oriented analysis and design, and the VisualAge product family.

Werner Frei teaches object-oriented analysis, design, and programming with Smalltalk and Java at the IBM education and training center in Switzerland. He has been with IBM since 1981, working in application development. Since 1991 he has focused on object technology, primarily as a consultant and mentor in VisualAge Smalltalk projects. In addition to teaching, Werner works on customer projects.

Bruno Georges is a member of the application development sales support team for IBM Europe, concentrating on object technologies (C++, Java, CORBA) and their supporting IBM products (VisualAge and Component Broker). His roles include customer consulting, assistance on pilot projects and proofs of technology, and presenting at conferences and shows. Bruno worked on the redbook, *World Wide Web Programming with VisualAge C++ and Smalltalk* (SG24-4734) and has consulted on the open systems architecture. He holds Master's degrees from the University of California at Berkeley and the Arts et Metiers Engineering school in Paris, France.

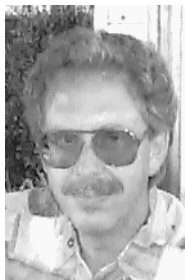
Paul Gover started as a programmer with IBM in 1975. He is now a Senior Application Development Specialist in the IBM Warwick development group, in Warwick, England. The group specializes in "turning good ideas into marketable products." Each member turns his or her hand to every aspect of application development, from architecture and design, through project management, code, test and documentation, to system test, packaging, and maintenance. Paul has worked with a broad range of programming languages but now specializes in object-oriented systems written in Smalltalk and Java. Paul holds a degree in Mathematics and a diploma in Computing from Cambridge, England.

Rudolf Wirawan is a consultant at the IBM Australian Programming Centre (APC) and for IBM banking customers in Australia and Hong Kong for client/server, SNA, TCP/IP, OS/2 database manager, object-oriented programming (Smalltalk, C++, and Java), distributed relational database architecture (DRDA), MQSeries, and cryptography. He specializes in application development and implementation of information system technology with information system (IS) and business strategic plans. Rudolf also specializes in people and organizational issues related to IS and IS business problems. Before joining IBM, he worked as a Senior System Software Designer at Wang Computer in Australia, as a System Support Manager at Nixdorf Computer in Indonesia, and as a System Software Engineer at Nixdorf Computer in Germany. He holds a degree in Mathematics from the University of Darmstadt, Germany.

Acknowledgments

This book would not have been possible without the help of the following people who contributed to the content of the book:

- ❑ **Marc Carrel-Billiard**, who wrote the first redbook on VisualAge for Java and helped me get to know the product in detail
- ❑ **Joaquin Picon** from the IBM Application Enabling Center of Competency in La Gaude, France, who wrote part of the chapter on Component Broker
- ❑ **Denise Hendriks** from the IBM development lab in Raleigh, North Carolina, and **Cynthia Dematteis** from IBM Argentina, who wrote the part on VisualAge Generator
- ❑ **Maggie Cutler**, who did her usual outstanding job at editing the book



U. Wahli

Ueli Wahli

Comments Welcome

Your comments are important to us!

We want our redbooks to be as helpful as possible. Please send us your comments about this or other redbooks in one of the following ways:

- ☐ Fax the evaluation form found in “ITSO Redbook Evaluation” on page 415 to the fax number shown on the form.
- ☐ Use the electronic evaluation form found on the Redbooks Web sites:

For Internet users <http://www.redbooks.ibm.com>

For IBM Intranet users <http://w3.itso.ibm.com>

- ☐ Send us a note at the following address:

redbook@us.ibm.com

1

Introducing VisualAge for Java Enterprise

In this chapter we describe the VisualAge for Java product family and introduce VisualAge for Java Enterprise. We cover all of the functions that come with the Enterprise product. We describe how the enterprise access builders work in general and explain in detail how each individual builder works.

We also describe how you can use the VisualAge for Java Enterprise product to access server applications that were developed with VisualAge Generator and Component Broker Connector.

We assume that you are familiar with the basic function of VisualAge for Java. For beginners we recommend the IBM Redbook entitled *Programming with VisualAge for Java*, IBM form number SG24-2232, published by Prentice Hall, ISBN 0-13-911371-1, April 1998.

VisualAge for Java Products

VisualAge for Java is available in three different versions:

- ❑ VisualAge for Java Professional
- ❑ VisualAge for Java Entry
- ❑ VisualAge for Java Enterprise

This book covers the VisualAge for Java Enterprise product without describing the basic functions of the VisualAge for Java Professional product.

VisualAge for Java Professional

The main feature of VisualAge for Java Professional is the integrated development environment (IDE), which enables you to create classes, or change methods, and then incrementally compile without the need to exit the testing phase of development. This IDE sports:

- ❑ A fully featured colored-syntax editor, which helps you write syntax-free source code. The source code is incrementally compiled when you save it.
- ❑ A debugger, which opens automatically when program exceptions occur
- ❑ Single-user version control, which lets you keep track of all changes you make to your code
- ❑ A Visual Composition Editor, which enables you to develop your application visually
- ❑ A JavaBeans creation tool to create 100% pure Java beans that you can use with the Visual Composition Editor

VisualAge for Java Entry

VisualAge for Java Entry has the same function as VisualAge for Java Professional, with a limit of 100 Java classes. You can get it for free from the Web (www.software.ibm.com/ad/vajava).

VisualAge for Java Enterprise

The main features of VisualAge for Java Enterprise are the enterprise access builders, which include:

- ❑ Data Access Builder for accessing enterprise data managed by database servers such as IBM's DB2 through the Java Database Connectivity (JDBC) specification for accessing relational databases from Java programs
- ❑ CICS Access Builder for building Java programs with external call interfaces (ECIs) to enterprise transactions managed by the IBM CICS Transaction Server for OS/390
- ❑ RMI Access Builder for building pure distributed Java applications that use the Java remote method invocation (RMI) specification
- ❑ C++ Access Builder for building distributed Java applications that connect to C++ application servers

In this book we describe, in detail, how to use the enterprise access builders to develop Java applications that access a multitude of servers.

Team Programming Feature

Also planned for the Enterprise product is a fully integrated, repository-based team programming feature that facilitates management of the development process on Java projects with complete source and version control.

The team programming feature is not available in the first shipment of the product; it will be added in the new version to be available in the third quarter of 1998.

Enterprise Access Builders

Figure 1 shows the general process of an enterprise access builder. The access builder analyzes the server specification and builds Java beans that encapsulate the public interface of the server.

The resulting Java beans are then used in VisualAge for Java to build applications. These beans are well suited for use in the Visual Composition Editor. Applications can be constructed by connecting graphical user interface (GUI) or controller features to the Java beans that encapsulate the original server.

The real server is used for testing and at execution time.

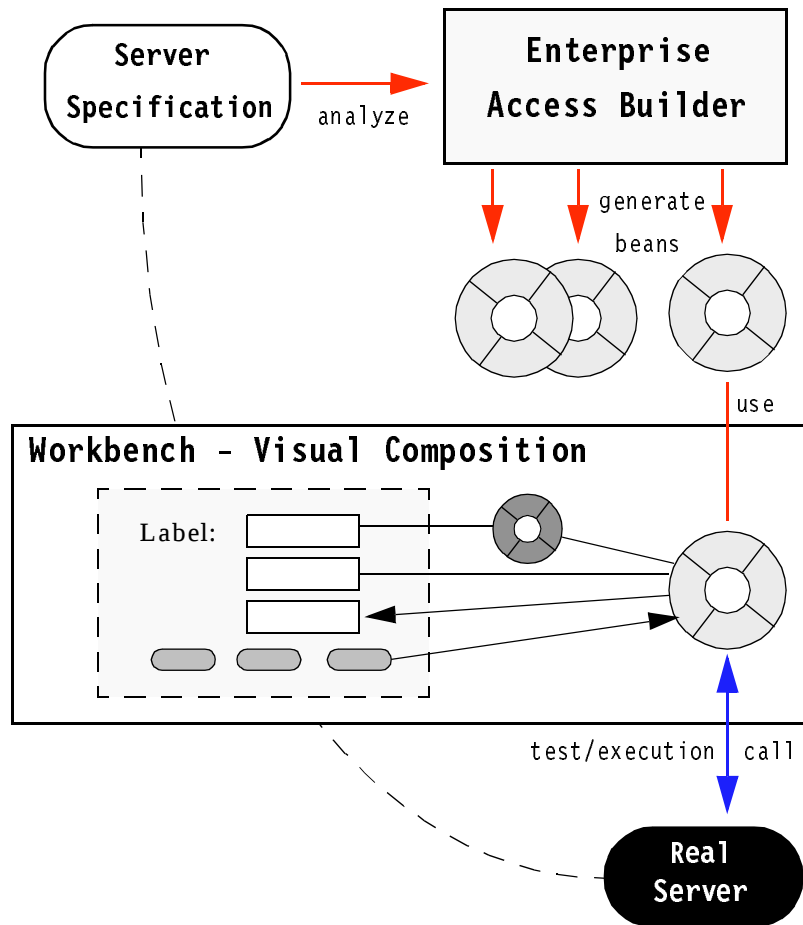


Figure 1. Development Process Using an Enterprise Access Builder

In general the four enterprise access builders work the same, but there are important differences:

- ❑ The Data Access Builder, the CICS Access Builder, and the RMI Access Builder are fully integrated with the VisualAge for Java Workbench. The Java beans that are generated by these builders are immediately stored in the repository and workspace.

Optionally the builders can run outside the Workbench, and the generated Java beans must be imported into the Workbench afterward.

- ❑ The C++ Access Builder is a command-line utility that produces C++ and Java source code. The C++ code must be compiled and the Java source code must be imported into the Workbench.

Figure 2 shows an overview of the four access builders.

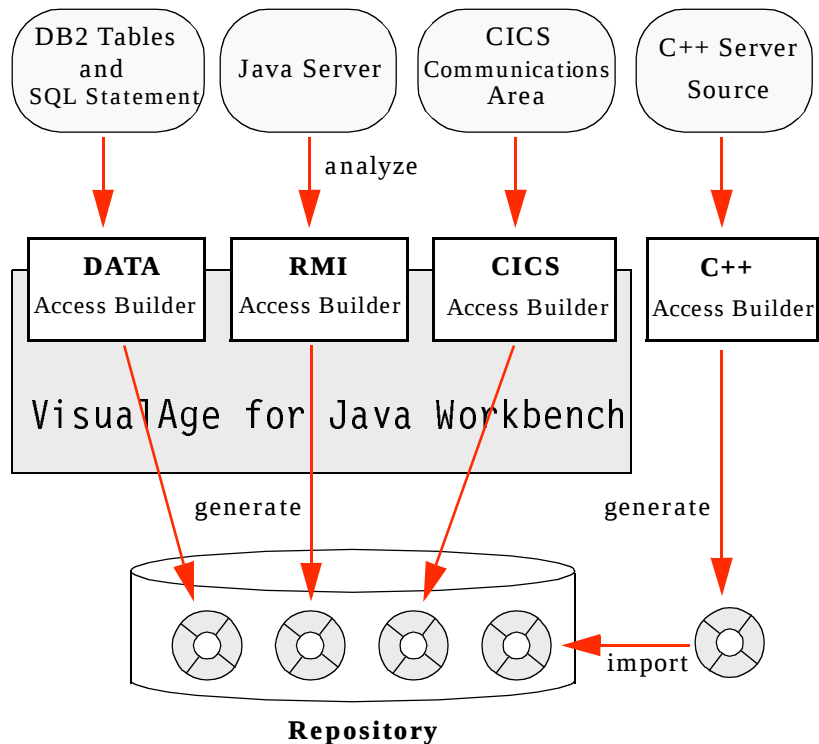


Figure 2. Enterprise Access Builder Overview

Data Access Builder Overview

The Data Access Builder is based on JDBC. Most vendors of relational databases provide drivers that implement the JDBC specification and allow Java applications to access their relational databases in three configurations (Figure 3).

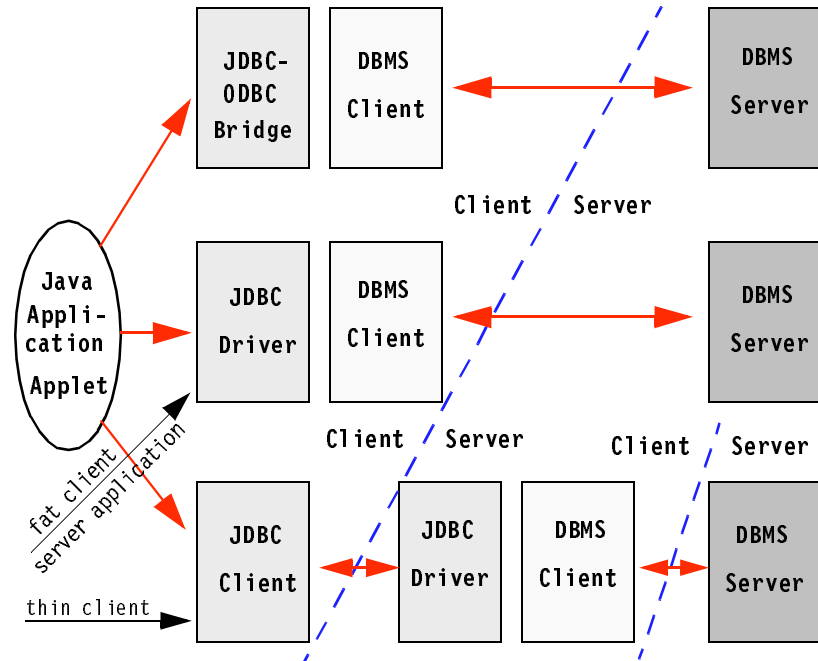


Figure 3. JDBC Database Access Configurations

The three configurations are:

- ❑ A JDBC-ODBC bridge that translates JDBC calls into matching ODBC calls that are forwarded to the database management system (DBMS).
- ❑ A JDBC driver that passes the requests to the DBMS. In this configuration the JDBC driver and at least one DBMS client must run on the same machine as the Java application. This is the normal configuration for server applications.
- ❑ A JDBC client that passes requests to a remote JDBC driver and the DBMS. This is a thin client configuration where no DBMS code is needed on the client machine. This is the best configuration for applets because the small JDBC client code can be downloaded with the applet.

In all three configurations it is possible to combine the DBMS client and server on the same machine.

Why a Data Access Builder?

The Data Access Builder encapsulates the JDBC code into Java beans. The client program accesses the beans without detailed knowledge of JDBC coding and database characteristics.

The Data Access Builder analyzes table or view definitions in the DBMS catalog and generates Java beans that encapsulate all access to the database tables into Java methods; that is, SQL select, insert, update, and delete statements are provided as methods of the generated Java beans.

The Data Access Builder also supports user defined SQL statements (for example, joins), stored procedures, and database operations such as connect, disconnect, and commit.

Figure 4 shows an overview of the Data Access Builder.

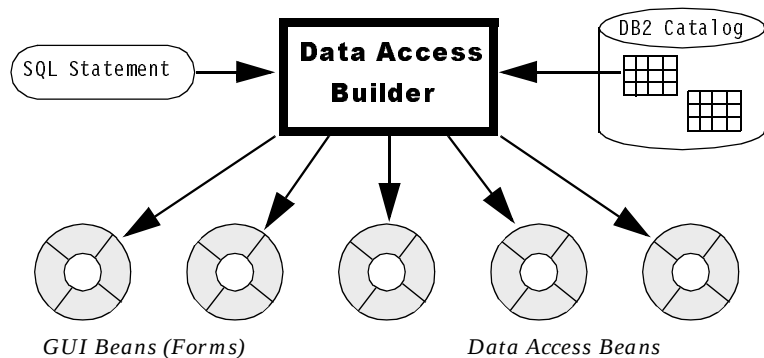


Figure 4. Data Access Builder Overview

The beans that are generated and imported into the repository of VisualAge for Java include:

- ☐ Single row access (retrieve, insert, update, delete)
- ☐ Multirow access (select)
- ☐ Database operations (connect, commit, rollback)
- ☐ Forms that can be used in a GUI
- ☐ An access application for direct database manipulation

The beans are then used to build applets and applications with the Visual Composition Editor. The developer is shielded from JDBC coding; all database access is available through the generated beans.

RMI Access Builder Overview

The RMI Access Builder is based on the RMI protocol of Java. RMI implements a Java client to Java server connection at an object level; that is, a local object can invoke a method in a remote object, pass objects as parameters, and get an object as a result. All communication is handled by the RMI system. Figure 5 shows an overview of an RMI client/server configuration.

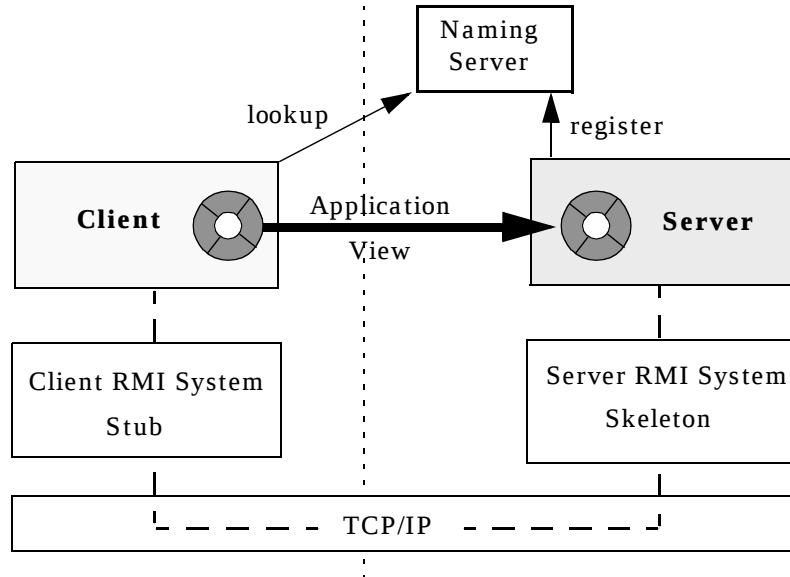


Figure 5. RMI Overview

Not much special code is involved in RMI programming:

- ❑ The RMI server must have its public interface defined, set up a security manager, and register with a naming server (RMI registry).
- ❑ The RMI client has to set up a security manager and look up the server in the remote naming server. A local proxy object is returned and can be used to invoke the remote methods.
- ❑ Communication between client and server is handled by two classes called *stub* and *skeleton*. These classes are generated by the RMI compiler from the server class.
- ❑ All objects that are passed as parameters to a remote method or returned from a method call must be serializable so that they can be sent through the network and restored in the other machine.

Why an RMI Access Builder?

For a simple application, direct coding of RMI is quite easy. For a more complex application, however, a lot of code must be written to propagate property values and events from the server to the client.

The RMI Access Builder makes the life of developers easy. Developers can concentrate on implementing the client and server, and the RMI Access Builder takes care of generating code for registration, lookup, and propagation of properties and events. The generated code is encapsulated in an RMI server proxy and RMI client proxy bean.

Figure 6 shows an overview of the RMI Access Builder.

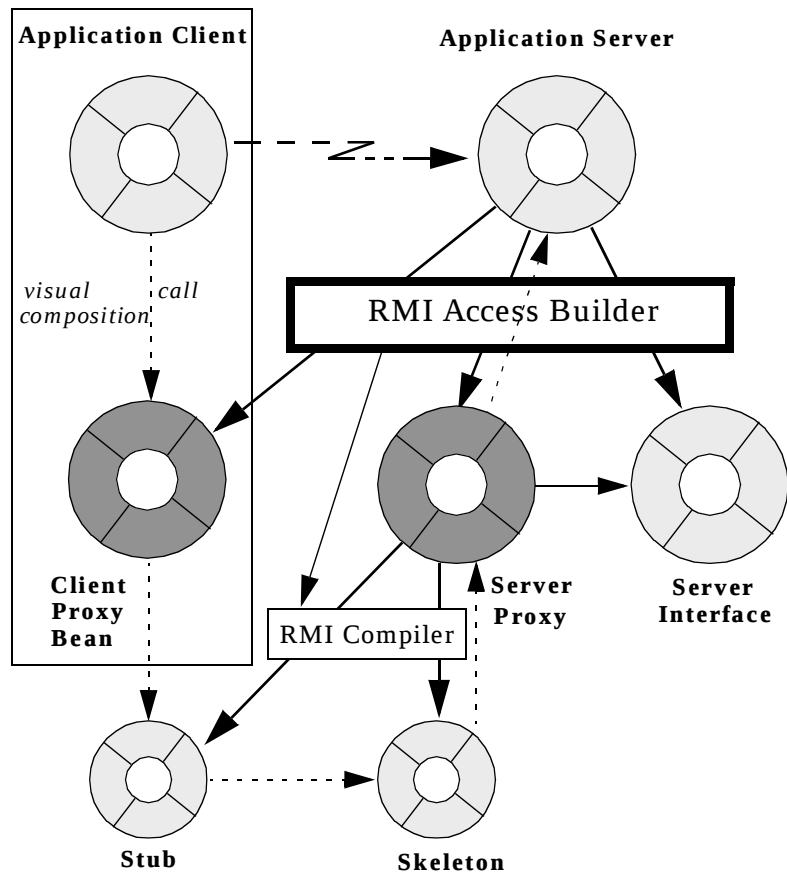


Figure 6. RMI Access Builder Overview

The RMI Access Builder analyzes the application server class and from its public interface generates:

- ❑ A server interface class that describes the public interface of the application server
- ❑ A server proxy that will be the real RMI server that will register itself with the naming server
- ❑ A client proxy bean that can be used in the Visual Composition Editor to build the client application. The proxy bean is a real Java bean that exposes the public interface of the server. The proxy bean looks up the proxy server and passes all remote requests to the server and results and events to the client.

The RMI Access Builder then calls the RMI compiler that generates the stub and skeleton classes from the proxy server class.

CICS Access Builder Overview

CICS provides a CICS Gateway for Java that enables Java applications to interact with a CICS server (Figure 7).

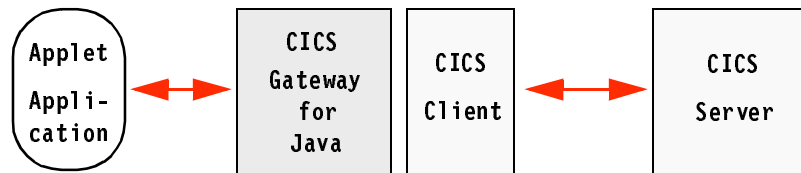


Figure 7. CICS Gateway for Java

The CICS Access Builder takes the specifications of a CICS server transaction and generates a Java bean that encapsulates all information required to invoke the transaction. The transaction specification must be in the form of a COBOL data structure of the CICS communications area that is passed to the transaction.

Helper classes are generated for marshaling of the properties of the communications area bean into a format suitable for transmission to and from the CICS Gateway for Java.

VisualAge for Java Enterprise provides a generic bean called a *CICS unit of work*, which is used to invoke the CICS transaction through the CICS Gateway for Java. The communications area bean is passed as a parameter to the CICS transaction.

Figure 8 shows an overview of the CICS Access Builder.

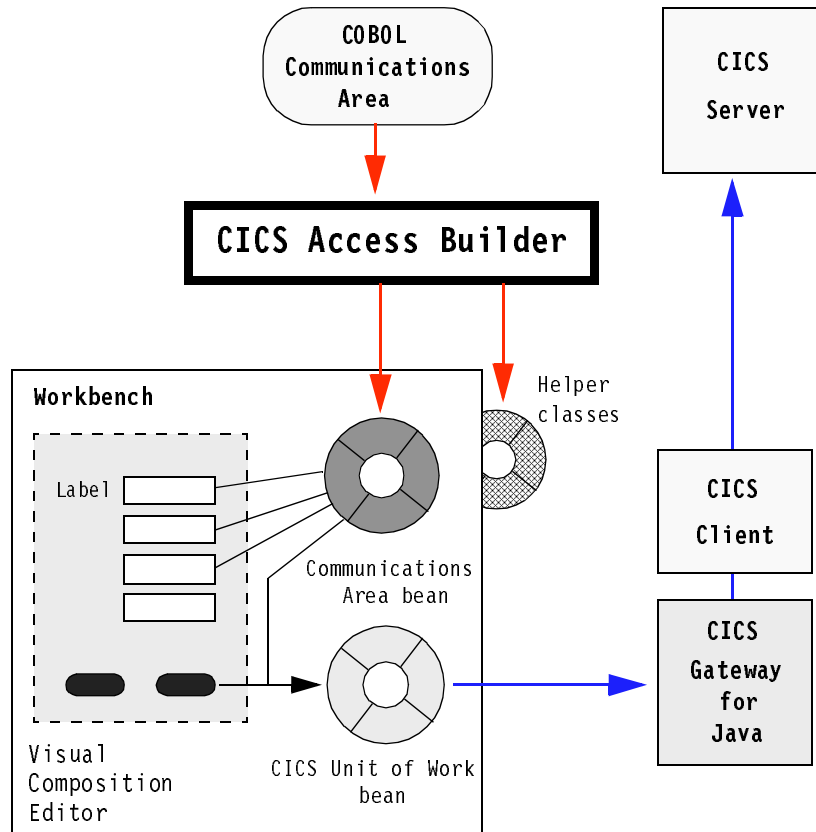


Figure 8. CICS Access Builder Overview

The communications area must be provided as a COBOL data structure or a full COBOL program. The CICS Access Builder creates a property in the communications area bean for each field in the COBOL data structure. Thus developers can use the Visual Composition Editor to connect the properties to GUI fields or other beans.

The CICS unit of work bean is provided by VisualAge for Java on the palette of the Visual Composition Editor. It is used to invoke the CICS transaction and for commit and rollback purposes. The communications area is passed as a parameter, and it is updated with changed values after the transaction is finished.

A helper class is generated for the communications area and for each contained substructure. The helper beans are used by the generated code automatically to translate the properties for communication.

C++ Access Builder Overview

The C++ Access Builder is based on the Java Native Method (JNI) specification. The JNI enables Java programs to invoke C and C++ functions. Coding of native method calls is quite complex and not suitable for many Java developers. In addition, existing C++ code might not adhere to the specifications of the JNI.

The C++ Access Builder analyzes existing C++ source code and generates:

- ❑ A C++ wrapper class that adheres to JNI specifications and passes the calls to the original C++ server
- ❑ A Java bean that can be used in Java applications to invoke the methods of the C++ server (through the wrapper class)
- ❑ A makefile that compiles and links the C++ wrapper class into a dynamic link library (DLL) that is loaded by the Java application at execution time

Figure 9 shows the C++ Access Builder process.

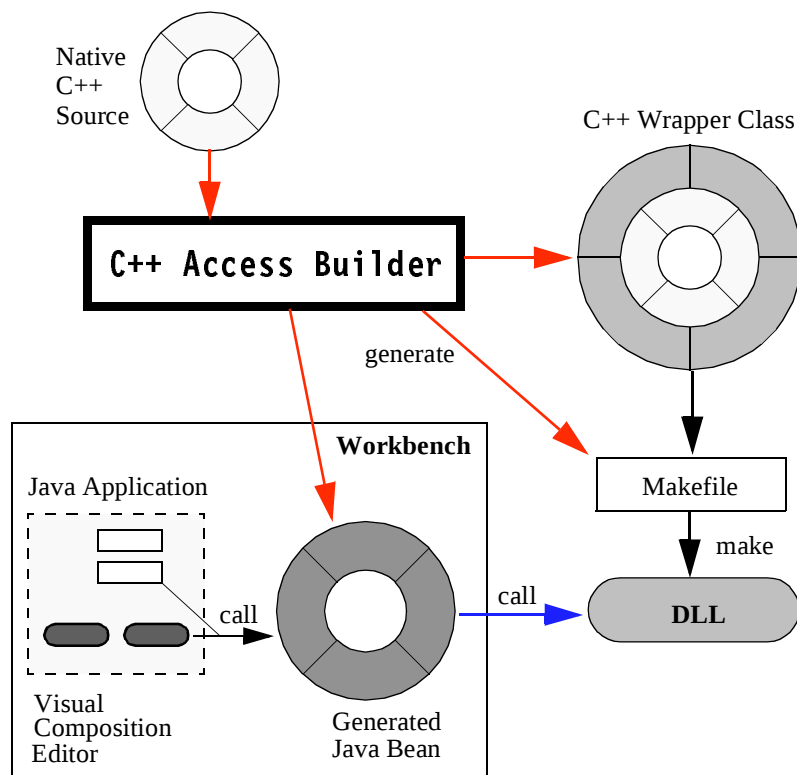


Figure 9. C++ Access Builder Overview

VisualAge for Java Enterprise Connectivity

In an applet or application, you can combine the different methods of connecting to enterprise servers in many ways.

Applet Connectivity

Figure 10 shows the connectivity for an applet. (We do not suggest using multiple different ways of connecting to enterprise servers in a single applet.)

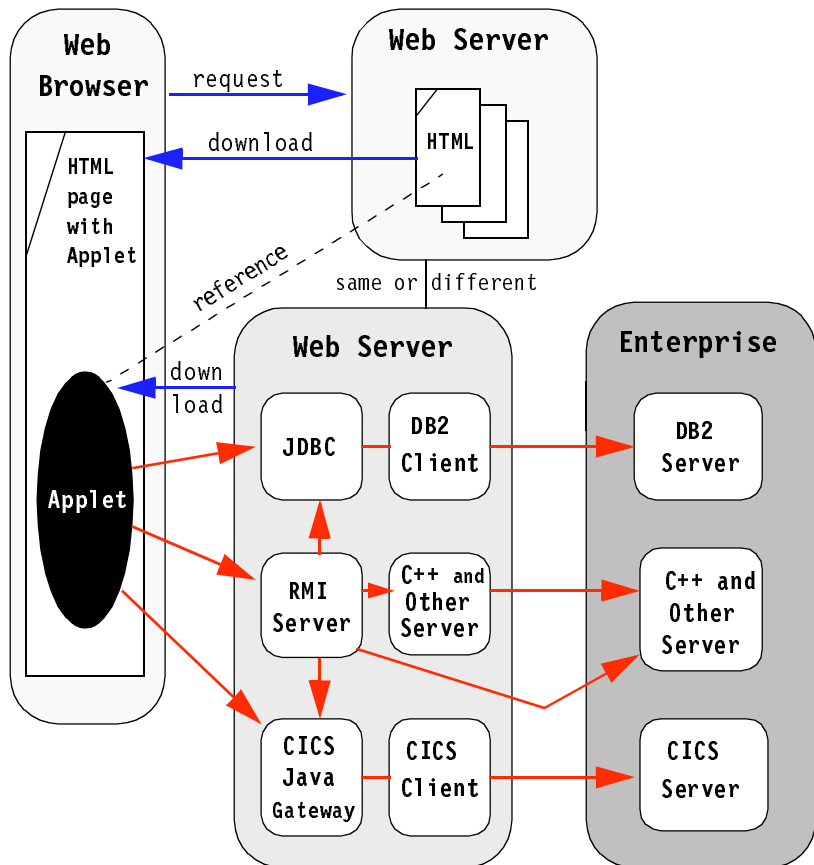


Figure 10. Applet Connectivity to Enterprise Servers

The RMI server is the most flexible of the servers. It is an application written in Java and therefore can connect to any other type of supported server.

Application Connectivity

Figure 11 shows the connectivity for applications.

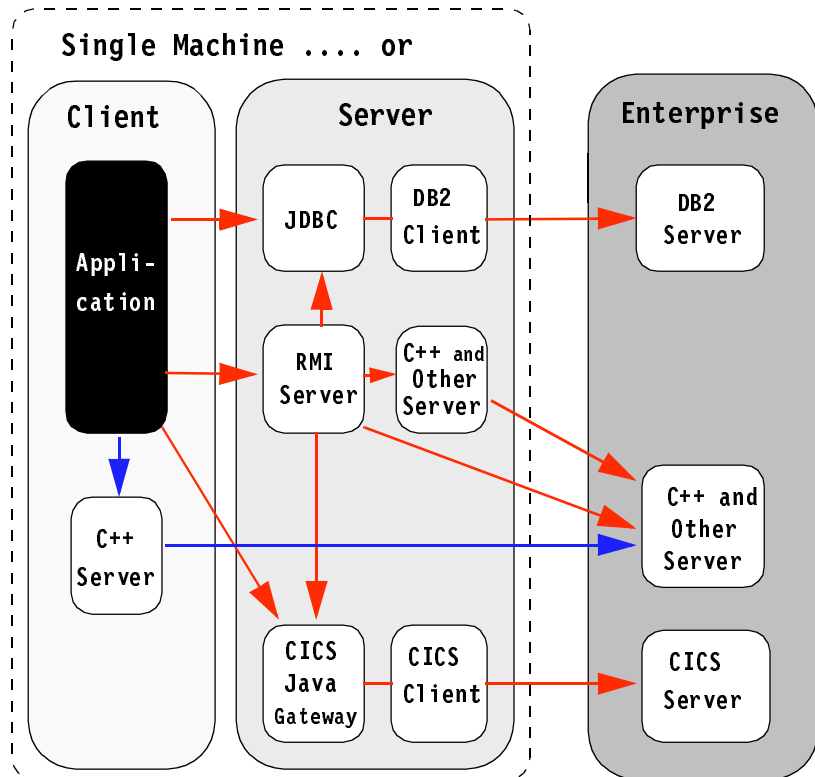


Figure 11. Application Connectivity to Enterprise Servers

An application does not have the restrictions that an applet has. The servers can be on the same machine or on another machine, except for the C++ server, which must be on the same machine.

Connecting to Other Servers

In addition to the four enterprise access builders of VisualAge for Java Enterprise, other products can provide Java beans to access servers from Java applications created with VisualAge for Java.

Connecting to VisualAge Generator

IBM's VisualAge Generator creates server applications and user interfaces for many platforms (OS/2, Windows, AIX, MVS, VM, VSE, and AS/400) from the same source specifications.

In its Version 2.2 with Fixpak and Version 3, VisualAge Generator creates Java beans that can be used in VisualAge for Java to create Java applets and applications that connect to VisualAge Generator servers.

Figure 12 shows an overview of a Java client with a VisualAge Generator Server. The process is similar to the enterprise access builders of VisualAge for Java:

- ❑ From the server source, VisualAge Generator generates a Java bean that represents the server and helper beans for the parameters that are used in calls to the server.
- ❑ VisualAge Generator generates the server program for the target platform.
- ❑ An applet (or application) is constructed with VisualAge for Java. The applet uses the server bean, the parameter beans, and a generic unit of work bean provided by VisualAge Generator.
- ❑ On the Web server, a gateway program provided by VisualAge Generator is started.
- ❑ To call the server, an applet invokes the unit of work bean, which communicates with the gateway through RMI. The gateway uses the VisualAge Generator PowerServer API and middleware to call the server on the target platform.
- ❑ A gateway program is not required for a Java application. However, the PowerServer API must be installed on the machine where the application runs.

In Chapter 10, "Access to VisualAge Generator Servers," on page 303, we describe the architecture and use of the VisualAge Generator support for Java.

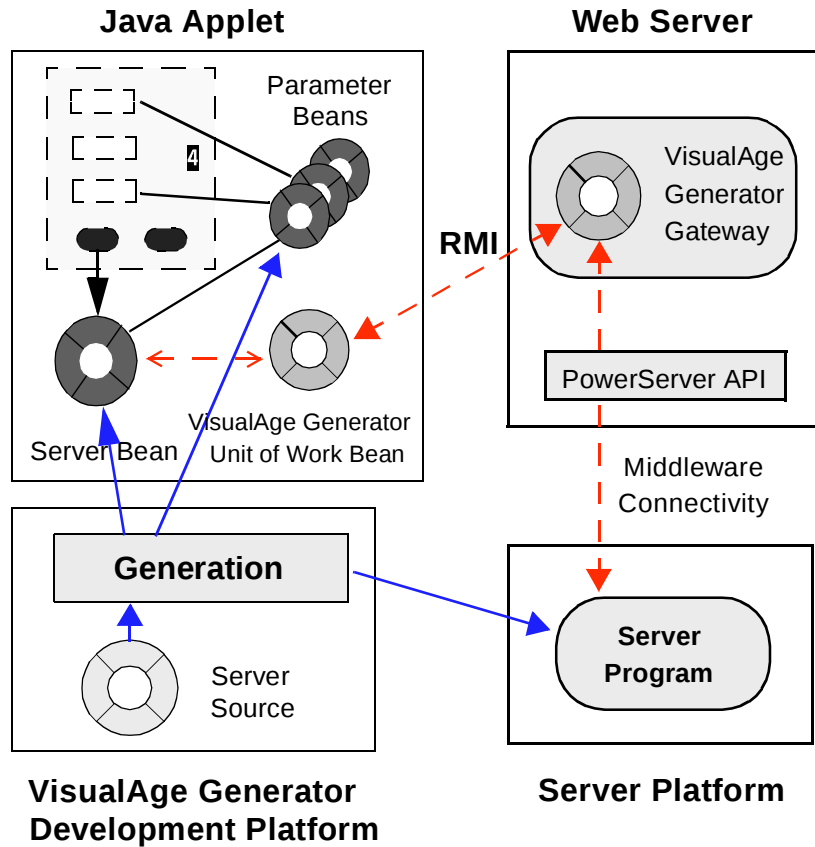


Figure 12. Java Applet with VisualAge Generator Server

Connecting to Component Broker

Component Broker is IBM's Common Object Request Broker Architecture (CORBA) implementation. It provides a development tool, CB Toolkit, and an execution environment, CBConnector, for C++ servers and clients written in C++, ActiveX, or Java.

CB Toolkit generates Java beans that can be used in VisualAge for Java to create client applications and applets that connect to CBConnector servers.

In Chapter 11, "Access to Distributed CORBA Objects," on page 317, we describe CORBA and Component Broker, and we develop a VisualAge for Java applet that connects to a CBConnector server.

2

Sample ATM Application and Database

In this chapter we introduce the sample ATM application that is implemented through the various facilities of VisualAge for Java Enterprise.

We list the application requirements and show a simple user interface design. We assume that the application is based on an existing relational database.

In subsequent chapters we use the enterprise access builders to implement the application with JDBC, RMI, CICS transactions, and C++ servers.

ATM Application Requirements

The ATM application handles two types of accounts: savings and checking. For both accounts, customers can perform debit and credit transactions. In addition, customers can list the transaction history belonging to an account.

Interest is credited to each savings account within a given period of time. Customers must maintain a minimum balance in a savings account and cannot withdraw funds beyond a specified overdraft amount from a checking account.

At the start of a business transaction, the ATM application prompts the customer to enter the ATM card identification number (card ID) on the Card ID panel. (This step simulates the ATM card reader installed at real ATM machines.)

On receiving a valid card ID, the ATM application greets the customer with the customer's name and title in the PIN panel. The card ID is redisplayed, so that the customer can verify the card. The ATM application prompts the customer for the personal identification number (PIN).

The ATM application verifies the PIN. If the PIN is invalid, a message is displayed indicating that the number is invalid, and the customer can reenter the PIN. On successful validation, a list of accounts that belong to the card is displayed in the Account panel.

The ATM application requests the customer to select an account for further processing. When the customer selects an account, the Transaction panel showing the customer information (title, name) and the account information (account ID, account type, old balance, new balance) is displayed.

At the start of a transaction, the new balance is the same as the old balance. The customer can enter an amount and proceed with a debit or a credit transaction. After every successful transaction, the new balance is displayed.

If the customer requests to see the account's transaction history, the ATM application displays the accumulated transactions in a drop-down list on the same panel.

From all panels the customer can return to the previous panel.

Figure 13 shows the basic layout of the panels and the application flow.

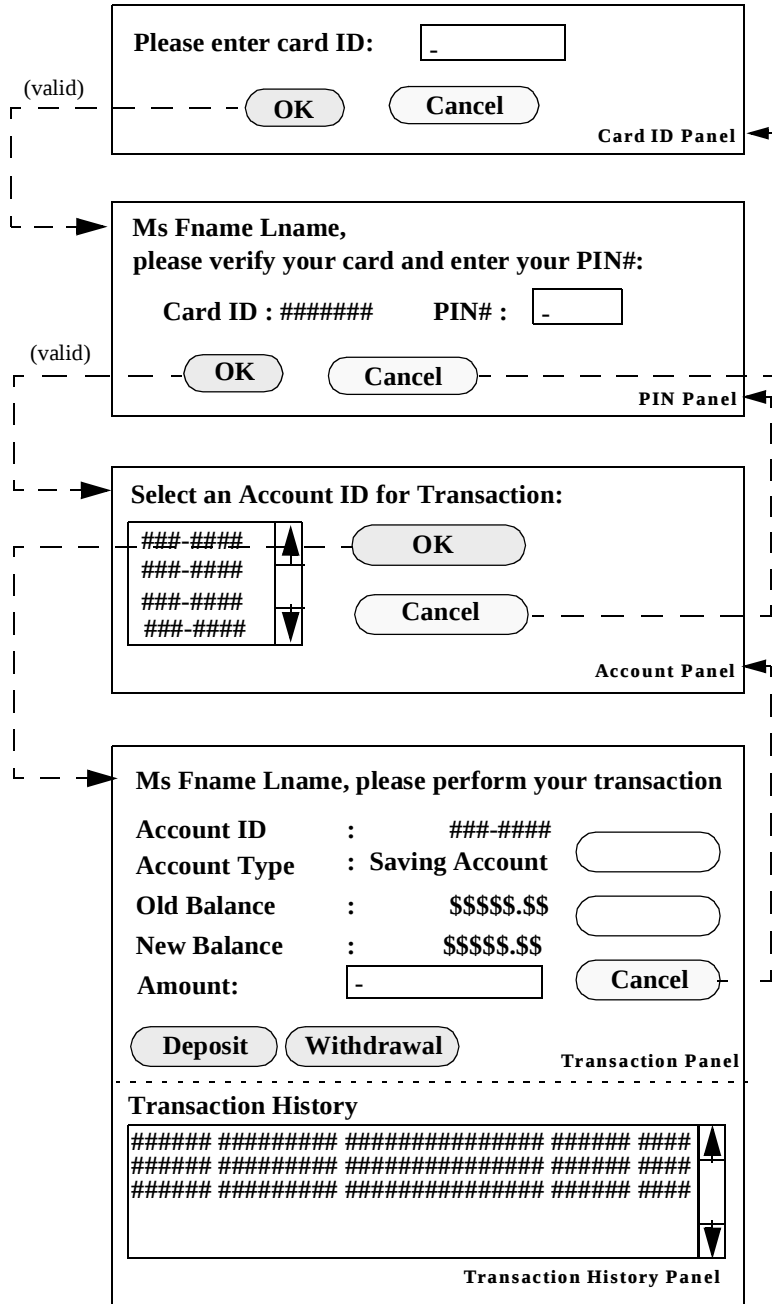


Figure 13. ATM Application Panels and Flow

Application Design

We consider the ATM application design in three steps:

1. User interface
2. Data store
3. Business logic

We will not implement all features specified in the ATM application requirements. To do so would increase the complexity of the programs and interfere with your understanding of and learning VisualAge for Java Enterprise.

We ignore the calculation and the capitalization of the interest for the savings account. Unlike a real ATM, our ATM continues to prompt the customer for a valid PIN, until the Cancel button is clicked.

User Interface

The user interface given in the ATM application requirements (see Figure 13 on page 19) gives us enough material to start with. The purpose of this book is to illustrate enterprise function, not to elaborate on user interface design.

Data Store

The ATM application has to remember the customer, the card, the account, and the transaction information. We use a relational database to store the information.

In "Database Design" on page 21, we describe the logical and physical database design of the ATM application, the implementation of the database, and the sample data for all ATM tables.

Business Logic

We have two type of accounts:

- ☐ Savings
- ☐ Checking

The main business logic functions are customer verification and account transactions. The following types of transactions have been identified:

- ☐ Pin validation
- ☐ Debit transaction
- ☐ Credit transaction

In addition, customer information, card information, and transaction history for each account have to be maintained.

Note: We do not describe the implementation of the business logic in this chapter. We will use different object models and approaches for the various enterprise access builders.

Database Design

We must make logical and physical database design decisions before we create the database.

In the logical database design, we describe the relationship between entities, and we list all attributes belonging to an entity. We also introduce the concept of referential constraints.

In the physical database design, we map each entity to a relational database table and each attribute of an entity to a column of a database table. We specify the type and the length of the columns in each table.

Logical Design

Our goal in designing the ATM database is to produce a representation of our environment that is easy to understand and will serve as a basis for expansion. In addition, we want a database design that will help us maintain consistency and integrity in our data. We can achieve these goals by producing a design that will reduce redundancy and eliminate anomalies that can occur during database updates.

Database design is not a linear process; in reality we probably have to redo steps as we work out the design.

We start by identifying the types of data to be stored in database tables. A database includes information about the entities in an organization or business and their relationships. In a relational database, entities are defined as tables.

The entity-relationship (ER) diagram shown in Figure 14 describes the ATM database.

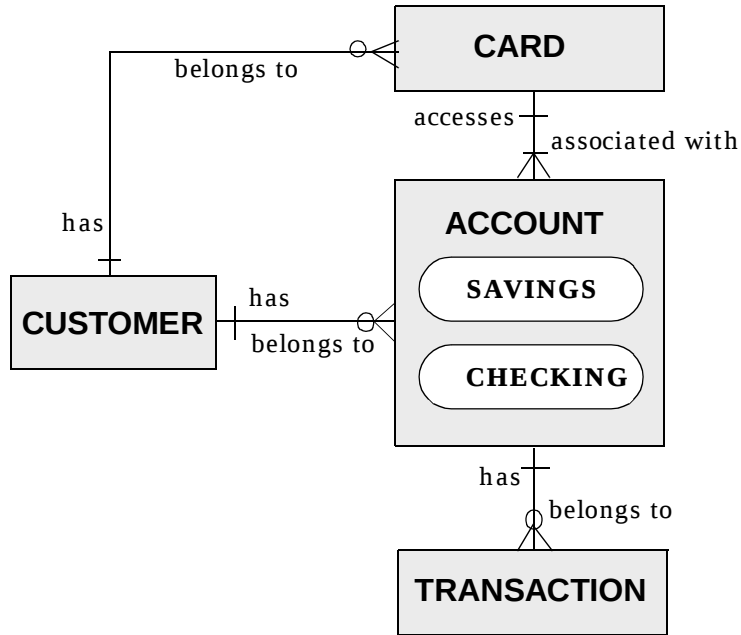


Figure 14. ATM Entity-Relationship Diagram

An entity is a person, object, or concept about which you want to store information. The entities described in the ATM tables are CUSTOMER, CARD, ACCOUNT, and TRANSACTION.

In the ATM design, the CUSTOMER entity has attributes, or properties, such as customer ID, title, first name, and last name. Those properties have the names CUSTID, TITLE, FNAME, and LNAME.

The CARD entity has two attributes, CARDID for the card ID, and PIN for the personal identification number.

The attributes of the ACCOUNT entity are ACCID for account ID, ACCTYPE for the account type, BALANCE for the account balance, MINAMT for the savings account minimum balance, and OVERDRAFT for the checking account overdraft amount. We added the ACCTYPE attribute to differentiate the savings account from the checking account.

The attributes of the TRANSACTION are TRANSID for the transaction ID, TRANSType for the transaction type, and TRANSAMT for the transaction amount.

Before we design the tables, we have to understand not only the entities but also their relationships.

The relationship between the CUSTOMER entity and the ACCOUNT entity is one-to-many. This can be expressed as a customer has zero or more accounts, and an account must belong to one customer.

The same logic applies to the CUSTOMER and CARD and to the CARD and ACCOUNT relationships. A customer has zero or many cards, but a card must belong to one customer. A card accesses one or many accounts, and an account must be associated with one card. These are also one-to-many relationships.

The relationship between the ACCOUNT entity and the TRANSACTION entity is one-to-many. This relationship can be expressed as an account has zero or more transactions, and each transaction must be identified by an account.

Reducing Redundancy and Eliminating Anomalies

To avoid redundancies and inconsistencies in the ATM data, we must normalize the ATM tables. The main idea in normalization is to reduce each table to a set of columns where all of the nonkey columns depend on the entire primary key of the table. If this is not the case, the data can become inconsistent during updates.

In fact, all tables in the ATM database are normalized at least up to the third normal form.

In this section we briefly review the rules for first, second, and third normal forms of tables. The fourth and the fifth normal forms of a table, which are covered in many books on database design, are not necessary for the ATM model and are not described here.

Here is a brief description of the normal forms:

- ❑ First normal form: In each cell position in the table, there is one and only one value.
- ❑ Second normal form: Each column that is not a key must depend on the entire key.
- ❑ Third normal form: Each nonkey column provides a fact that is independent of other nonkey columns and dependent only on the key.

Figure 15 depicts a situation that satisfies the second normal form, but not the third normal form. The dependency between *nonkeyN* and the *key* is transitive; therefore *nonkeyN* is dependent on the *key*, which satisfies the second normal form but not the third.

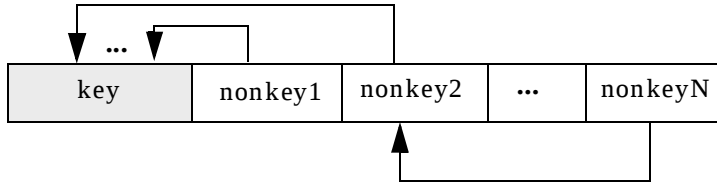


Figure 15. Table with Third Normal Form Violation

In Figure 16, all *nonkeyX* attributes are dependent on the *key* attribute. This satisfies not only the second normal form but also the third normal form.

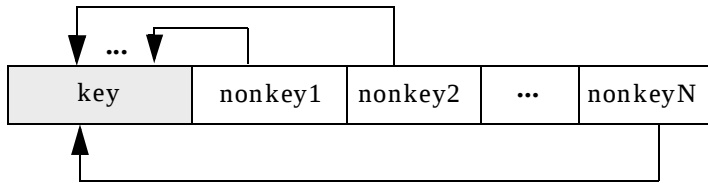


Figure 16. Table in Third Normal Form

Referential Integrity

Referential integrity lets you define required relationships between and within tables. The database manager maintains these relationships, which are expressed as referential constraints, and requires that all values of a given attribute or column of a table also exist in some other table or column. For example, a typical referential constraint might require that every account in the ACCOUNT table must belong to a customer that exists in the CUSTOMER table. No account can exist without an owner.

You can build referential constraints into a database to ensure that referential integrity is maintained. When planning for referential integrity, identify the relationships to be established between database tables. You can identify a relationship by defining a primary key and referential constraints.

Figure 17 depicts the referential integrity constraints of the ATM tables.

Customer:

CUSTID (PK)	TITLE	FNAME	LNAME
----------------	-------	-------	-------

Card:

CARDID (PK)	PIN	CUSTID
----------------	-----	--------

Account:

ACCID (PK)	CARDID (FK)	CUSTID (FK)	ACCTYPE	BALANCE	MINAMT	OVERDRAF
---------------	----------------	----------------	---------	---------	--------	----------

Transaction:

TRANSID (PK)	ACCID (FK)	TRANSTYPE	TRANSAMT
-----------------	---------------	-----------	----------

Referential constraints:
On delete restrict

Figure 17. Referential Integrity Constraints in the ATM Database

The following definitions are useful for understanding referential integrity:

- ❑ A primary key is a column (or set of columns) that provides a unique identification for each row of the table. Each table can have only one primary key. In Figure 17, the CUSTID and ACCID tables are the primary keys of the CUSTOMER and ACCOUNT tables.
- ❑ A foreign key is a column (or set of columns) in a table that refers to a unique or primary key of the same or another table. A foreign key is used to establish a relationship with a primary key to enforce referential integrity among tables. In Figure 17 on page 25 the CARDID column in the ACCOUNT table is a foreign key because it refers to the primary key, column CARDID, of the CARD table.
- ❑ A parent key is a primary key or unique key of a referential constraint.

- ❑ A parent table is a table containing a parent key that is related to at least one foreign key in the same or another table. A table can be a parent in an arbitrary number of relationships. For example, in Figure 17 the CUSTOMER table, which has a primary key of CUSTID, is a parent of the ACCOUNT table, which contains foreign key CUSTID.
- ❑ A dependent table is a table containing one or more foreign keys. A dependent table can also be a parent table. A table can be a dependent in an arbitrary number of relationships. For example, in Figure 17 the ACCOUNT table contains the foreign key, CUSTID, which is dependent on the CUSTOMER table, which has a primary key. In turn the ACCOUNT table is also the parent table of the TRANSACTION table.
- ❑ A referential constraint is an assertion that non-null values of a designated foreign key are valid only if they also appear as values of the unique key of a designated table. The purpose of referential constraints is to guarantee that database relationships are maintained and data entry rules are followed.

There are three referential integrity rules:

- ❑ Insert
- ❑ Delete
- ❑ Update

Insert Rules

You can insert a row at any time into a parent table without any action being taken in the dependent table. However, you cannot insert a row into a dependent table unless there is a row in the parent table with a unique key value equal to the foreign key value of the row that is being inserted, unless the foreign key is null.

Delete Rules

When you delete a row from a parent table, the database manager checks whether there are any rows in the dependent table with matching foreign key values. If any dependent rows are found, several actions can be taken. You specify which action is taken by specifying a delete rule when you create the dependent table.

The delete rules for a dependent table, when a primary key is deleted, are:

- ❑ Restrict: Prevent any row in the parent table from being deleted if any dependent rows are found. If you need to remove both parent and dependent rows, delete the dependent rows first.

- ❑ Cascade: Implies that deleting a row in the parent table automatically deletes any related rows in the dependent table. This rule is useful when a row in the dependent table has no significance without a row in the parent table.
- ❑ Set null: Ensures that deletion of a row in the parent table sets the values of the foreign key in any dependent rows to null. Other columns of the dependent row are unchanged.

Update Rules

The database manager prevents the update of a unique key of a parent row. When you update a foreign key in a dependent table, and the foreign key is not null, it must match an existing value of the parent key in the parent table of the relationship.

Applying Referential Integrity to the ATM Application

By applying referential integrity to the ATM application, the following data consistency can be achieved:

- ❑ The delete rule for the ATM application has been set to *Restrict*. This prevents anyone from deleting a row in the parent table while dependent rows exist.
- ❑ The insert rule prevents the situation where a new account is created without an association with a customer. Such an account is called an *orphan* or *phantom* account.
- ❑ The update rule prevents anyone from reassociating an account with a nonexistent customer.

Physical Design

After we have completed the logical database design but before implementation, we must map each entity to a table in the relational database and each attribute to a column of a table.

In the physical design, we specify the type of the columns, the length of the data, the primary keys, and if null values are allowed.

Tables 1 through 4 show the physical design of the ATM tables.

Table 1. Customer Table

Column Name	Type	Length	Key	Nulls	Description
CUSTID	CHAR	4	Yes	No	Customer ID
TITLE	CHAR	3	No	No	Title
FNAME	CHAR	30	No	No	First name
LNAME	CHAR	30	No	No	Last name

Table 2. Card Table

Column Name	Type	Length	Key	Nulls	Description
CARDID	CHAR	7	Yes	No	Card ID
PIN	CHAR	4	No	No	PIN
CUSTID	CHAR	4	No	No	Customer ID

Table 3. Account Table

Column Name	Type	Length	Key	Nulls	Description
ACCID	CHAR	8	Yes	No	Account ID
CARDID	CHAR	7	No	No	Card ID
CUSTID	CHAR	4	No	No	Customer ID
ACCTYPE	CHAR	1	No	No	Account type (S = Savings, C = Checking)
BALANCE	DEC	(8, 2)	No	No	Balance
MINAMT	DEC	(8, 2)	No	No	Minimum amount
OVERDRAF	DEC	(8, 2)	No	No	Overdraft amount

Table 4. Transaction Table

Column Name	Type	Length	Key	Nulls	Description
TRANSID	TIME-STAMP	26	Yes	No	Transaction ID
ACCID	CHAR	4	No	No	Account ID
TRANSTYPE	CHAR	1	No	No	Transaction type (D = Debit, C = Credit, T = Transfer)
TRANSAMT	DEC	(8, 2)	No	No	Transaction amount

ATM Database

After determining the database design, we use command line processor commands and SQL statements to create the database and the objects within it. These objects can include schemas, node groups, table spaces, tables, views, aliases, user-defined types (UDTs), user-defined functions (UDFs), triggers, constraints, indexes, and packages.

Figure 18 shows the data definition language we used to implement the ATM database.

```

echo --- connect to ATM database ---
CONNECT TO ATM

echo --- creating tables ---
CREATE TABLE ATM.CUSTOMER (
    custid    CHAR( 4) NOT NULL PRIMARY KEY, \
    title     CHAR( 3) NOT NULL,           \
    fname     CHAR(30) NOT NULL,           \
    lname     CHAR(30) NOT NULL            \
)
CREATE TABLE ATM.CARD (
    cardid    CHAR(7) NOT NULL PRIMARY KEY, \
    pin       CHAR(4) NOT NULL,             \
    custid    CHAR(4) NOT NULL,             \
    FOREIGN KEY (custid) REFERENCES ATM.CUSTOMER ON DELETE RESTRICT \
)
CREATE TABLE ATM.ACCOUNT (
    accid     CHAR(8)  NOT NULL PRIMARY KEY, \
    cardid    CHAR(7)  NOT NULL,             \
    custid    CHAR(4)  NOT NULL,             \
    acctype   CHAR(1)  NOT NULL,             \
    balance   DEC(8,2),                      \
    minamt    DEC(8,2),                      \
    overdraf  DEC(8,2),                      \
    FOREIGN KEY (custid) REFERENCES ATM.CUSTOMER ON DELETE RESTRICT, \
    FOREIGN KEY (cardid) REFERENCES ATM.CARD ON DELETE RESTRICT \
)
CREATE TABLE ATM.TRANS (
    transid   TIMESTAMP NOT NULL PRIMARY KEY, \
    accid     CHAR(8)   NOT NULL,             \
    transtype CHAR(1)   NOT NULL,             \
    transamt  DEC(8,2)  NOT NULL,             \
    FOREIGN KEY (accid) REFERENCES ATM.ACCOUNT ON DELETE RESTRICT \
)
echo --- execute GRANT statements ---
GRANT BINDADD ON DATABASE TO PUBLIC
GRANT CONNECT ON DATABASE TO PUBLIC
GRANT ALL ON ATM.CUSTOMER TO PUBLIC
...

echo --- connect reset ---
CONNECT RESET

```

Figure 18. ATM Database Data Definition Language

Sample Data of ATM Tables

The sample data of the ATM tables shows the internal relationships among the ATM tables:

- ❑ We have six customers, with numbers 101 to 106.
- ❑ There are seven ATM cards with numbers 1111111 to 7777777, and matching PINs 1111 to 7777.
- ❑ Account numbers are structured xxx-yyyy, where xxx is the customer number.

Tables 5 through 8 list the sample data of the ATM tables.

Table 5. Customer Table Sample Data

CUSTID	TITLE	FNAME	LNAME
101	Sig	Stefania	Celentano
102	Mr	Rudolf	Wirawan
103	CH.	Werner	Frei
104	Sir	Paul	Gover
105	Mon	Bruno	Georges
106	ITS	Ueli	Wahli

Table 6. Card Table Sample Data

CARDID	PIN	CUSTID
1111111	1111	101
2222222	2222	102
.....		
6666666	6666	106
7777777	7777	106

Table 7. Account Table Sample Data

ACCID	CARD-ID	CUST-ID	ACC-TYPE	BALANCE	MIN-AMT	OVERDRAF
101-1001	1111111	101	C	80.00	0.00	100.00
101-1002	1111111	101	C	195.22	0.00	400.00
101-1003	1111111	101	S	9375.26	100.00	0.00
102-2001	2222222	102	S	19375.26	9999.99	0.00
....						
106-6666	6666666	106	C	6.66	0.00	0.00
106-7777	7777777	106	S	111.11	11.11	0.00

Table 8. Transaction Table Sample Data

TRANSID	ACCID	TRANSTYPE	TRANSAMT
1997-10-07-14.30.26.720001	101-1001	C	80.00
CURRENT TIMESTAMP	101-1002	C	200.00
...			
CURRENT TIMESTAMP	106-7777	D	111.11

3

Java Database Connectivity

In this chapter we introduce the JDBC concepts and show how Java applications can access relational databases. We describe the different categories in which JDBC drivers are organized.

We implement an application and an applet that, through JDBC drivers, connect to and retrieve data from local and remote databases.

Database Application Architectures

To access a vendor-specific database, you use a vendor-specific database engine, whose SQL implementation is in many cases incompatible with another vendor's implementation. Incompatibilities occur mainly in the areas of embedded SQL and stored procedures.

To resolve incompatibilities you have to write two different applications if you have to support two different databases (Figure 19).

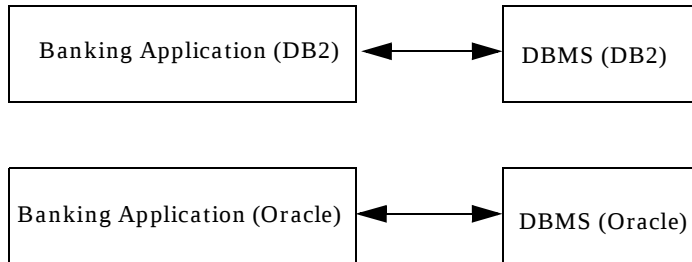


Figure 19. A Banking Application with Two Different Databases

The Ideal Solution for Programmers

You can avoid writing applications to support two different databases by adding data access layers consisting of a generic data access part and a DBMS-specific data access part (Figure 20).

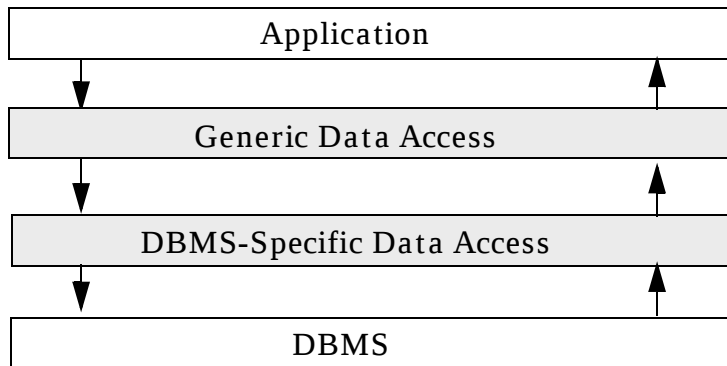


Figure 20. Data Access Layers: Generic and DBMS-Specific

One implementation of such an architecture is Open Database Connectivity (ODBC) as shown in Figure 21.

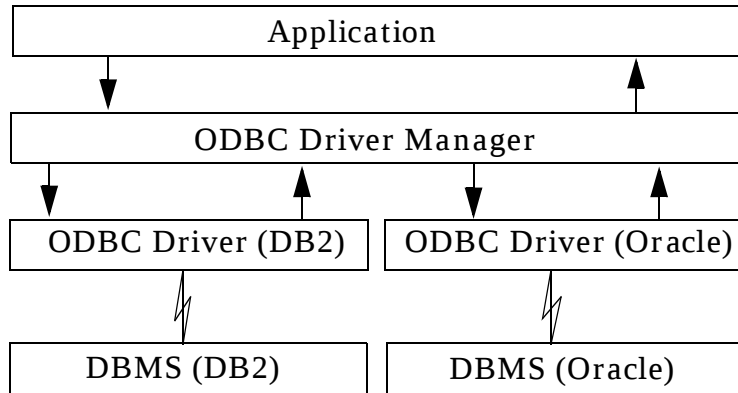


Figure 21. Open Database Connectivity Architecture

Another situation with incompatibilities occurs if your application has to support databases on two different platforms. Many programming languages, for example, C, C++, and COBOL, are not fully portable between platforms. To avoid this situation, you can write your application in an interpretive language such as Java.

Using the ideal solution for both data access and language support, you can now write one application regardless of the database type and platform that it must support. Having the same Java language and the same database application programming interface (API) on all platforms solves the programmer's problem. The Java interpreter handles the data access layer of the database engine and the platform dependent issues. However, to optimize the utilization of the database, you need some sort of client/server configuration.

Single-Tier Architecture

In the early days of database application development, a monolithic architecture, also known as a single-tier architecture, was used. There is no network and the application and the data store reside on the same machine (Figure 22).

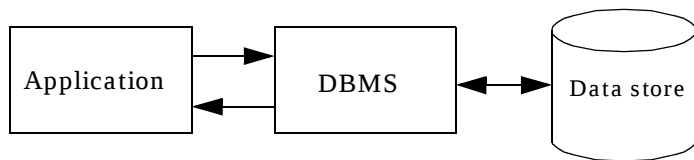


Figure 22. Monolithic (Single-Tier) Architecture

Most of the applications written in those days were rather simple. For larger applications the model did not scale and the applications were tightly coupled to the DBMS. Furthermore, only one instance of the application could access the DBMS.

Two-Tier Architecture

In a two-tier architecture, the DBMS is divided into two components, the client and the server (Figure 23). The application and the DBMS' client component can be installed in a machine other than the DBMS server.

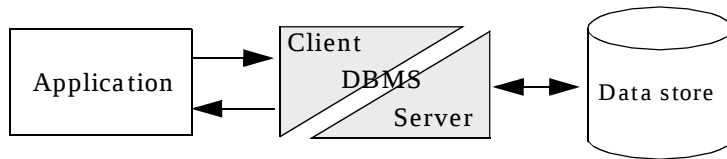


Figure 23. Client/Server (Two-Tier) Architecture

With this architecture, more than one instance of an application can access the DBMS server and therefore optimize the utilization of the database (Figure 24).

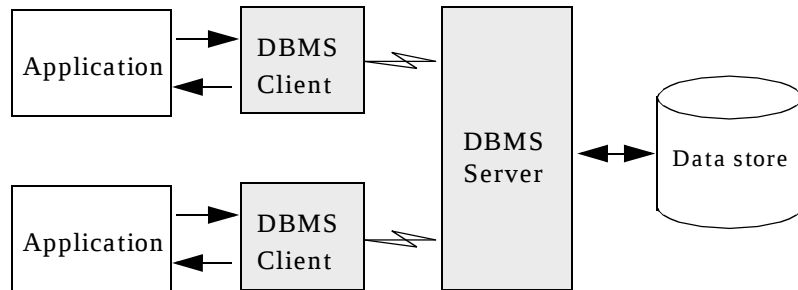


Figure 24. Multiple Client Instances Accessing One Server

As more and more application logic is built into the client, the client itself becomes large and ineffective. The many clients are hard to maintain because they must have a DBMS component installed. This type of client is a *fat client*.

Three-Tier Architecture

Most of the database applications written in the two-tier architecture do not use the solution we described in “The Ideal Solution for Programmers” on page 34 and are therefore DBMS dependent. The application has to be changed if it has to support another DBMS. To avoid this situation, you must divide the application into two parts: DBMS-dependent code and DBMS-independent, reusable, code (Figure 25).



Figure 25. Dividing the Application into Client and Server

Using the client/server architecture, you can place the DBMS-dependent portion into the application server and the DBMS-independent portion into the application client. In this architecture, you only have to write one client application to access different types of DBMSs. The client machine becomes thin and easy to manage. This type of client is a *thin client*, and the architecture is called a three-tier architecture (Figure 26).

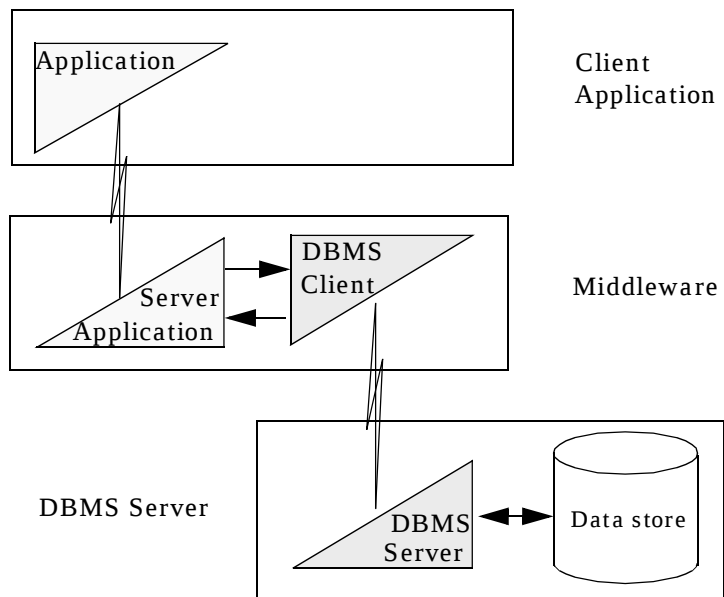


Figure 26. Three-Tier Architecture

In intranet and Internet environments, the Web browser is the first tier, the Web server is the second tier, and the DBMS server is the third tier. The secondary tier is also referred to as the middleware.

Using a three-tier architecture, you can build your application and divide it into two components. The first component, for example, a Java applet, is the client application; it can be run in the client machine, after it has been requested by the Web browser.

The second component, which has all the business logic, remains on the Web server. This application component has a connection to the DBMS client to service all applets that are connected to the Web server.

In the sections that follow, we describe the data access layer based on a two-tier architecture, which simplifies the explanation and applies in most cases to a three-tier architecture (see Figure 20 on page 34).

JDBC

In a two-tier architecture, the client application uses the data access layer to communicate with the server. The communication between a client and a server is through a network protocol. The server resides on a system other than the client (Figure 27).

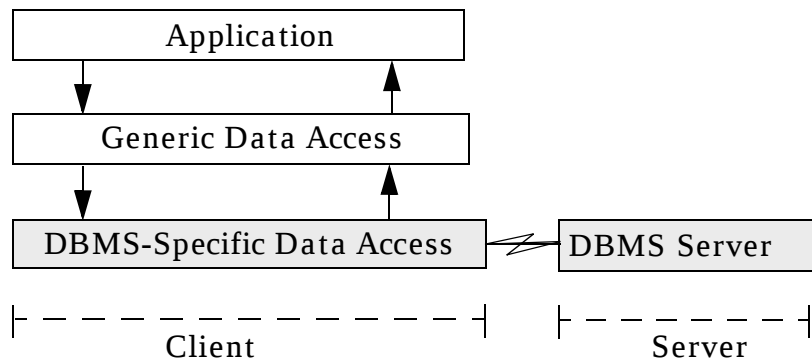


Figure 27. Data Access Layer Revised

Let us now look at how JavaSoft, the Sun Microsystems Business Unit responsible for the development of Java products, implements the data access layer.

JDBC Drivers

JavaSoft implemented the data access layer similar to Microsoft's ODBC API, which is probably the most widely used API for accessing relational databases in Windows operating systems, and called its implementation *JDBC*.

The Structure of an Ideal JDBC Driver

Before we describe the structure of an ideal JDBC driver, let us consider the Internet and intranet environment in Figure 28.

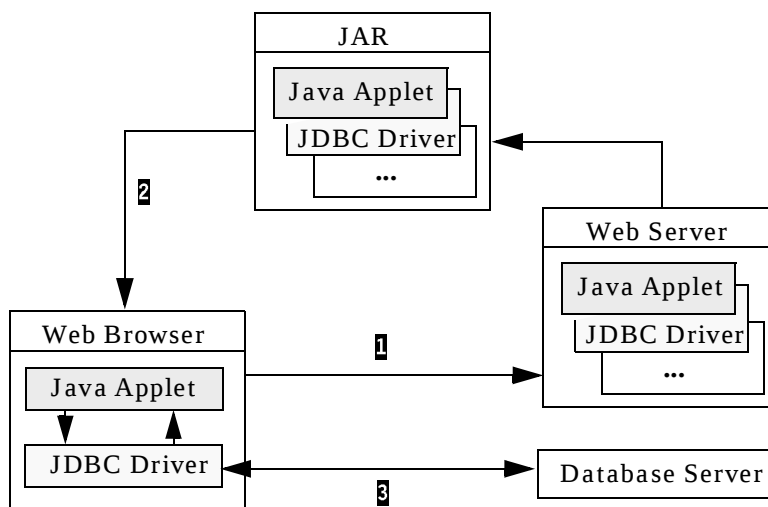


Figure 28. JDBC Applet in the Intranet and Internet Environment

You use your favorite Web browser (client), point to a URL, and request to run a database application from a Web server (1).

The server responds by sending the requested applet (the database application), including the JDBC driver and other required resources, bundled into a jar file to the Web browser (2).

As soon as the jar file has been successfully downloaded, the Web browser starts the applet, which uses the JDBC driver to access the database (3).

In this scenario, no additional program and configuration are required in the client machine, because the JDBC driver is automatically shipped with the applet in a jar file. The only software required in the client is the Web browser, and we have a thin client configuration.

An ideal JDBC driver is a database driver written entirely in Java and used by an application to access a database. The database server responds directly to the JDBC driver requests without any additional interface.

The communication between the JDBC driver and the database server is through a network protocol, which must be built into the database engine. The JDBC driver talks directly to the database through Java sockets (Figure 29).

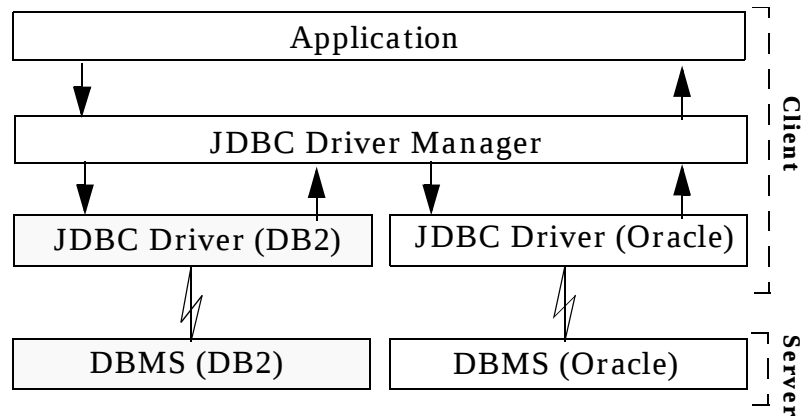


Figure 29. An Ideal JDBC Driver

This type of driver is provided in most cases by the database vendor. JavaSoft has categorized this type of driver as a category 4 driver.

In “Building on Existing Products” we describe the drivers of category 1, 2, and 3 as interim solutions.

An application using the ideal JDBC API interfaces directly with the JDBC driver manager. The JDBC driver manager in turn loads the JDBC driver that handles access to the requested database.

This is the ideal solution. It will take some time for all database vendors to implement it, however, because the database driver has to be written in Java, and the network protocol must be built into the database engine.

Building on Existing Products

To be competitive and responsive to customer demand, vendors capitalize on existing products, so that their products can be released to market quickly.

JavaSoft has identified three interim solutions for developing a JDBC driver for an existing product:

- ❑ JDBC and ODBC bridge driver
- ❑ JDBC and vendor-specific bridge driver
- ❑ JDBC generic network protocol driver

JDBC and ODBC Bridge Driver

JavaSoft provides a bridge between Java applications and ODBC drivers to utilize the many existing ODBC drivers for different database engines (Figure 30).

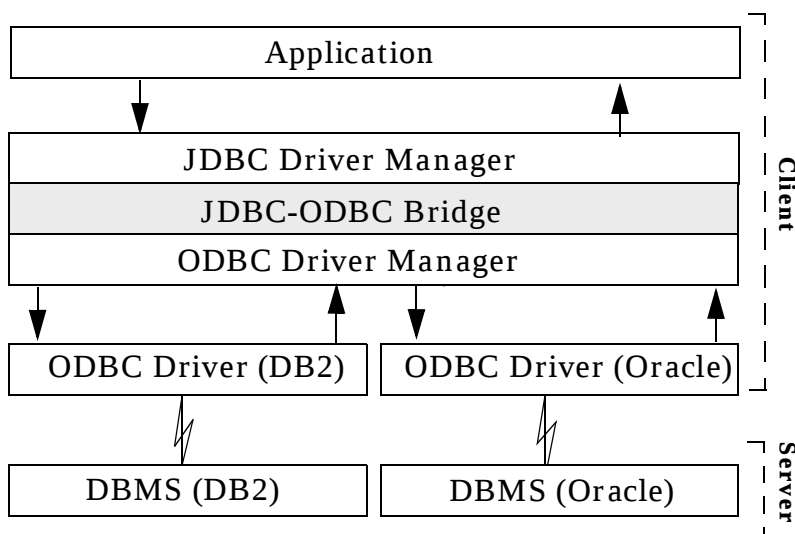


Figure 30. JDBC-ODBC Bridge Driver

There is a very close relationship between the ODBC (Figure 21 on page 35) and JDBC (Figure 29 on page 40) architectures and APIs. Both ODBC and JDBC are based on the X/Open SQL Command Level Interface.

JavaSoft categorizes the ODBC Bridge driver as a category 1 driver.

JDBC and Vendor-Specific Bridge Driver

ODBC is not the only way to access a database. Most database vendors already have their own drivers to access their databases.

IBM, for example, uses the DB2 Client Application Enabler (CAE) driver to access the DB2 database server (Figure 31).

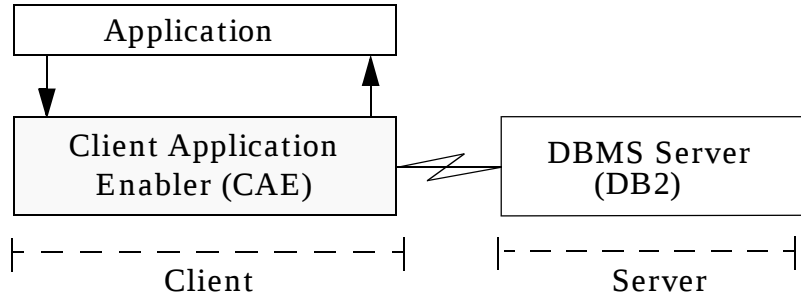


Figure 31. Vendor-Specific Driver

To make use of this type of driver in a Java environment, a JDBC vendor-specific bridge is required (Figure 32).

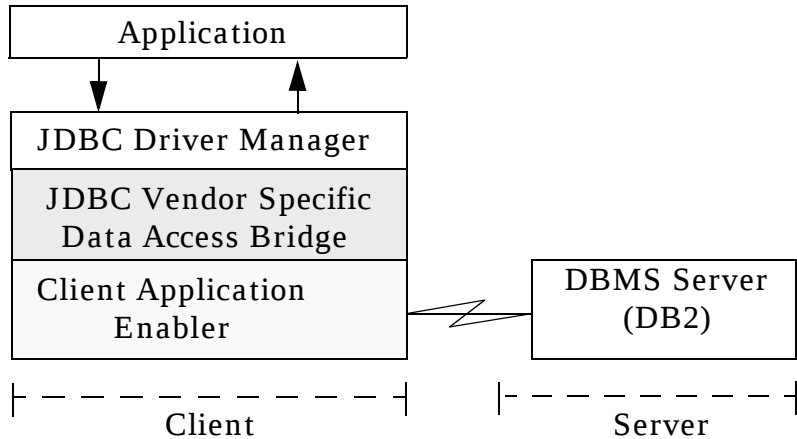


Figure 32. JDBC Vendor Specific Bridge

JavaSoft categorizes this type of driver as a category 2 driver.

JDBC Generic Network Protocol Driver

The category 1 and category 2 drivers cannot be used in an Internet environment, because both driver managers depend on a set of libraries written in a language other than Java. This is true for the ODBC bridge driver and vendor-specific drivers, such as DB2 CAE.

To solve this problem, we divide the ideal JDBC driver (Figure 29 on page 40) into two components, a client component and a server component, and move all non-Java language functions to the server component, so that the client portion can be written in “pure” Java language.

The client portion is responsible for translating the JDBC calls into a database-independent network protocol, and the server component is responsible for translating the database-independent network protocol into database calls. The server component is also referred to as the *middleware* component (Figure 33).

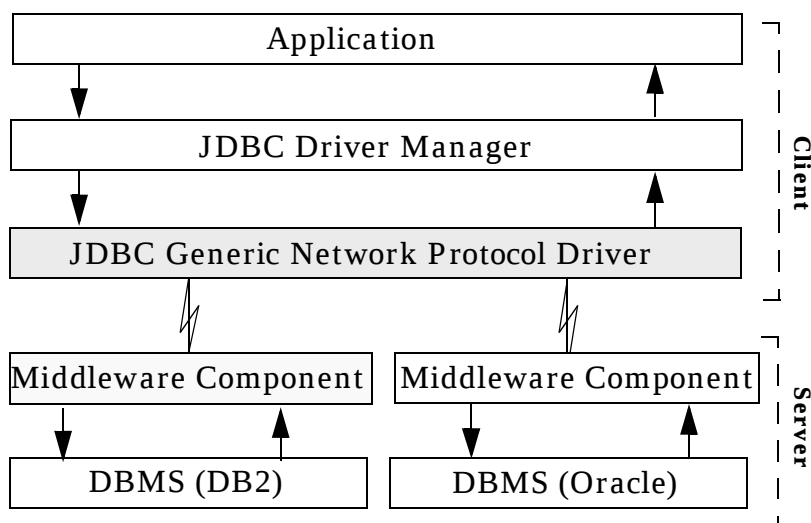


Figure 33. JDBC Generic Network Protocol Driver

The JDBC generic network protocol driver is extremely flexible because it does not require code installed on the client, and a single driver can provide access to multiple databases.

JavaSoft categorizes this type of driver as a category 3 driver.

Now that we understand the various ways to connect with JDBC to a database, we can explain how a Java database application is structured.

Structure of a JDBC Application

Figure 34 shows the structure of a JDBC application. Before you can communicate with a database, you must load the related JDBC driver. Using the `DriverManager` class, you can load the driver and then make a connection to the database.

After the connection is successful, you create an instance of a statement class. The statement object is used to represent the SQL statement. There are three types of statements:

- ☐ Statement (without host variables)
- ☐ Prepared statement (with place holders for host variables)
- ☐ Callable statement (stored procedure)

The result set object manages the rows retrieved by an SQL select. The result set maintains the position of the current row. The next method moves to the next data row.

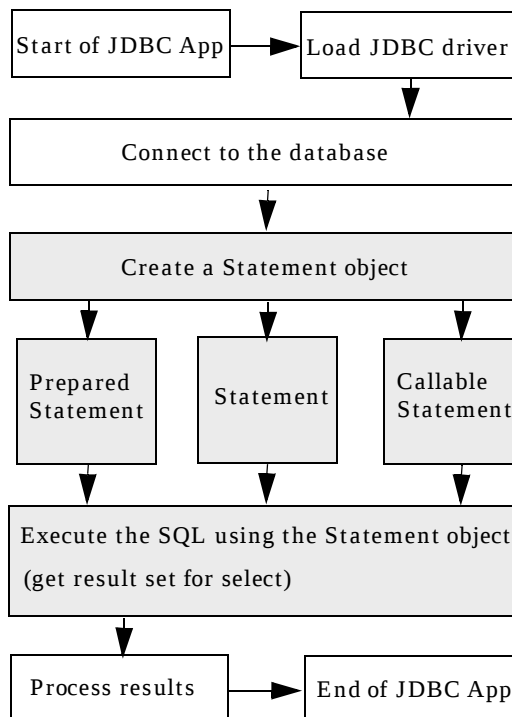


Figure 34. Structure of JDBC Application

JDBC provides an interface that enables you to determine the type of data returned, so that you can get information about the data itself (meta data).

For a given result set object, an application can call the `getMetaData` method to retrieve descriptive meta data (for example, the number of columns, their names, and data types).

JDBC Connection Sample

In this section we present a “minimal” JDBC program to illustrate how to connect to a database. This program has been written as simply as possible, similar to “Hello, World,” so that you can create a minimal functional JDBC program to test your connection (Figure 35).

To connect to any database you must load a JDBC driver. DB2 provides two types of drivers: category 2 (vendor-specific bridge) and category 3 (generic network protocol). We use both drivers to implement a local and a remote database connection.

For category 2, you have to install the DB2 CAE, a native DB2 driver, on the client machine. For category 3, you have to start a JDBC server daemon on the server machine.

The DB2 JDBC drivers are denoted—in their full class name—by:

- ❑ `COM.ibm.db2.jdbc.app.DB2Driver` (category 2)
- ❑ `COM.ibm.db2.jdbc.net.DB2Driver` (category 3)

To load the driver, an application uses the Java Class method `forName(String)`, which returns the Class object associated with the class of the given string name. The `newInstance` method creates an actual object for the loaded class. The `newInstance` method is required on certain platforms (OS/2) and works on all platforms (for example, Windows 95 and Windows NT):

```
Class.forName("COM.ibm.db2.jdbc.app.DB2Driver").newInstance();
```

The `DriverManager` attempts to load the class with the given name through the current `CLASSPATH`. If the requested driver is not found, the `DriverManager` throws a `ClassNotFoundException` exception.

To make the sample as simple as possible, we do not specifically catch the `ClassNotFoundException` but more generally catch every exception that can occur in this program.

After you have successfully loaded the driver, you connect to the database, using a method provided by the `DriverManager`:

```
getConnection(String url, String user, String password);
```

If the program runs successfully, the Connected to database message is displayed on the console.

Otherwise you could see:

```
java.sql.SQLException: [IBM][CLI Driver] SQL1032N
                        no start database manager command was issued
```

```
java.lang.ClassNotFoundException: COM.ibm.db2.jdbc.app.DB2DriverXYZ
```

Figure 35 shows the complete sample program.

```
import java.sql.*;
public class JdbcConnect
{
    public static void main(String[] args)
    {
        try
        {
            // load the JDBC driver with DriverManager
            Class.forName("COM.ibm.db2.jdbc.app.DB2Driver").
                newInstance();
            // Identify the data source
            String url = "jdbc:db2:sample";
            // Allocate a Connection object
            Connection con = DriverManager.getConnection(url,
                                                         "userid", "password");
            System.out.print("Connected to database\n");
            // Close the Connection object
            con.close();
        }
        catch (Exception e)
        { System.out.println(e); }
    }
}
```

Figure 35. Sample JDBC Program

This sample JDBC program uses the DB2 category 2 driver (COM.ibm.db2.jdbc.app.DB2Driver) to connect to a local database.

To connect to a remote database, you can use the DB2 category 3 driver (COM.ibm.db2.jdbc.net.DB2Driver). In the getConnection method, you specify the location of your database as a JDBC URL:

```
jdbc:db2://IPhostname:port/databaseName
```

There is no default port number for JDBC; we use 8888. To connect to a remote DB2 database through a category 3 driver, you start the DB2 JDBC server daemon before trying the connection:

```
DB2JSTRT 8888
```

The general syntax of the JDBC URL is:

```
JDBC:<subprotocol>:<subname>
```

The <subprotocol> identifies which driver to use, and <subname> provides the driver with any required connection information.

For example, to access a database, you can use the following connection:

```
jdbc:db2:databaseName (local)
jdbc:db2://serverHostname:8888/databaseName (remote)
```

In summary, to access a remote database you have to change two statements in the program:

```
Class.forName("COM.ibm.db2.jdbc.net.DB2Driver").newInstance();
String url = "jdbc:db2://serverHostname:8888/sample";
```

JDBC Applications and JDBC Applets

From a JDBC programming structure point of view, there are no significant differences between applications and applets, other than the fact that an applet must use a category 3 driver, whereas an application can use either a category 2 or 3 driver.

The example that follows is built on the JdbcConnect sample (Figure 35). We implement the program as both a JDBC application and a JDBC applet.

The program retrieves data from the ORG table in the DB2 sample database. The result of the program should be similar to the result when you execute the SQL statement shown in Figure 36 in a DB2 command window.

```
DB2CLP C:\SQLLIB\bin>db2 select * from org
```

DEPTNUMB	DEPTNAME	MANAGER	DIVISION	LOCATION
10	Head Office	160	Corporate	New York
15	New England	50	Eastern	Boston
20	Mid Atlantic	10	Eastern	Washington
38	South Atlantic	30	Eastern	Atlanta
42	Great Lakes	100	Midwest	Chicago
51	Plains	140	Midwest	Dallas
66	Pacific	270	Western	San Francisco
84	Mountain	290	Western	Denver

```
8 record(s) selected.
```

Figure 36. DB2 Sample Organization Table

JDBC Application

In this application program (Figure 37), we use the category 3 driver, COM.ibm.db2.jdbc.net.DB2Driver, to access the database remotely.

```
import java.sql.*;
public class SampleOrg
{
    public static void main(String[] args)
    {
        try
        {
            int n, i = 0;
            StringBuffer aString;
            String aDriverName = "COM.ibm.db2.jdbc.net.DB2Driver";
            Class.forName(aDriverName).newInstance();
            String url = "jdbc:db2://senegal:8888/sample";
            Connection con = DriverManager.getConnection(url, "stade1", "stade1");
            // Allocate a Statement object
            Statement stmt = con.createStatement();
            // Execute a query using the Statement object
            ResultSet rs = stmt.executeQuery("SELECT * FROM ORG");
            ResultSetMetaData rsmd = rs.getMetaData();
            int nrColumns = rsmd.getColumnCount();
            System.out.print("\n");
            System.out.println("DEPT    DEPTNAME          MGR    DIVISION    LOCATION");
            System.out.println("-----");
            // Retrieve data from the returned ResultSet object
            while (rs.next())
            {
                for (n=1; n <= nrColumns; n++)
                {
                    aString = new StringBuffer(rs.getString(n));
                    aString.setLength(rsmd.getColumnDisplaySize(n) + 1);
                    System.out.print(aString);
                }
                System.out.print("\n");
                i++;
            }
            System.out.println("\n" + i + " record(s) selected");
            // Close the ResultSet and Statement object
            rs.close();
            stmt.close();
            con.close();
        }
        catch (ClassNotFoundException e)
        { System.out.println(e); }
        catch (SQLException sqle)
        { System.out.println(sqle); }
        catch (InstantiationException ie)
        { System.out.println(ie.toString()); }
        catch (IllegalAccessException iae)
        { System.out.println(iae.toString()); }
    }
}
```

Figure 37. JDBC DB2 Organization Application

As you can see, the beginning of the program is the same as the `JdbcConnect` class example (Figure 35 on page 46), except for the driver name and the URL. You can, of course, continue to use the category 2 driver (`COM.ibm.db2.jdbc.app.DB2Driver`) if you want.

In this example, *senegal* is the host name of the remote machine, and *stadel* is the user ID and password. For tests with a local database on your own machine you can use the *loopback* or *localhost* host name and your own user ID and password.

Notice that we handle the `ClassNotFoundException` and the `SQLException` exceptions. We also have to handle the `InstantiationException` and the `IllegalAccessException` exceptions, because they could be thrown by the `newInstance` method. The `newInstance` method has been included, so the program can be run in non-Windows environments as well.

We need an SQL statement to select, insert, update, or delete one or more rows in a table. Using the `Connection` object, we can create a `java.sql.Statement`:

```
Statement stmt = con.createStatement();
```

A `Statement` object enables us to execute a select statement that returns a `ResultSet` object:

```
ResultSet rs = stmt.executeQuery("SELECT * FROM ORG");
```

A `ResultSet` maintains a cursor pointing to its current row of data. Initially the cursor is positioned before the first row. Using the `next` method of the `ResultSet`, we can access the next row.

```
rs.next();
```

Using a `ResultSetMetaData` object we can analyze the types and properties of the columns in a `ResultSet`:

```
ResultSetMetaData rsmd = rs.getMetaData();
```

In this program, we use the `getColumnCount` method to get the number of the columns and the `getColumnDisplaySize` method to get the size of each column. The `getString` method returns the value of a column in the current row as a Java string. The `StringBuffer`, in conjunction with the `getColumnDisplaySize` method, permits us to format the rows retrieved.

To end the program, we close the `ResultSet` object, the `Statement` object, and the `Connection` object, as shown in the example.

To run with remote database access we start the DB2 JDBC daemon on the server (`DB2JSTRT 8888`).

JDBC Applet

Our JDBC applet is pretty much the same as our JDBC application. For simplicity, we retrieve only the first record from the ORG table and use only the init and paint methods (Figure 38).

```
import java.awt.*;
import java.sql.*;
public class OrgApplet extends java.applet.Applet
{
    String depnum, depname, manager, division, location;
    public void init()
    { try
      {
          String aDriverName = "COM.ibm.db2.jdbc.net.DB2Driver";
          Class.forName(aDriverName).newInstance();
          String url = "jdbc:db2://senegal:8888/sample";
          Connection con = DriverManager.getConnection(url, "stade1", "stade1");
          Statement stmt = con.createStatement();
          ResultSet rs = stmt.executeQuery("SELECT * FROM ORG WHERE DEPTNUMB=10");
          // Retrieve the first record from SAMPLE database
          rs.next();
          depnum = rs.getString(1);           // retrieve by column number
          depname = rs.getString("DEPTNAME"); // retrieve by column name
          manager = rs.getString(3);
          division = rs.getString("DIVISION");
          location = rs.getString(5);
          rs.close();
          stmt.close();
          con.close();
      }
      catch (Exception e) { System.out.println(e); }
    }

    public void paint(Graphics g) {
        g.drawString("Department Number", 25, 30);
        g.drawString(": "+depnum, 150, 30);
        g.drawString("Department Name", 25, 50);
        g.drawString(": "+depname, 150, 50);
        g.drawString("Manager", 25, 70);
        g.drawString(": "+manager, 150, 70);
        g.drawString("Division", 25, 90);
        g.drawString(": "+division, 150, 90);
        g.drawString("Location", 25, 110);
        g.drawString(": "+location, 150, 110);
    }
}
```

Figure 38. JDBC DB2 Organization Applet

To run this program, you need an HTML file. For example:

```
<HTML> <title> JDBC Applet Tester </title>
<center> <h1> JDBC APPLET TESTER </h1> </center>
The record from the ORG table:
<p>
<applet code="OrgApplet.class" width=500 height=120> </applet>
</HTML>
```

Running the HTML file with the applet viewer

```
appletviewer OrgApplet.html
```

produces the results shown in Figure 39.



Figure 39. JDBC Applet in the AppletViewer

JDBC Sample for Insert, Update, and Delete

Figure 40 shows you how to write a Java program with SQL insert, update, and delete statements. For the purpose of simplicity, we only manipulate one row of the Organization table (Figure 36 on page 47). In contrast to the previous JDBC application (SampleOrg.java in Figure 37 on page 48), we do not format the rows retrieved from the table.

We first insert one record with the department number 11, update the department name, and then delete the record, so that we can run it many times.

In each process, we list the content of the ORG table before we change anything, and we list it again after modifications.

Note the use of `getInt` and `getFloat` methods to retrieve column values from the result set object. There is a matching `get` method for each data type. DB2 converts a numeric column value to the data type of the Java variable.

```

import java.sql.*;
public class JDBCTest
{
    Connection con;
    Statement stmt;
    public static void main(String[] args)
    { JDBCTest SQLTest;
      try
      {
          SQLTest = new JDBCTest();
          System.out.println("--- ORG List Before Modifications ---");
          SQLTest.orgList();
          System.out.println("--- SQL Add DEPTNUMB 11 ---");
          SQLTest.insert();
          SQLTest.orgList();
          System.out.println("--- SQL Update DEPTNAME ---");
          SQLTest.update();
          SQLTest.orgList();
          System.out.println("--- SQL Delete DEPTNUMB 11 ---");
          SQLTest.delete();
          SQLTest.orgList();
      }
      catch(SQLException sqle) { System.out.println("SQL Exception: " + sqle); }
      catch(ClassNotFoundException cnfe) { System.out.println(cnfe.toString()); }
      catch(InstantiationException ie) { System.out.println(ie.toString()); }
      catch(IllegalAccessException iae) { System.out.println(iae.toString()); }
    }
    public JDBCTest() throws SQLException, ClassNotFoundException,
        InstantiationException, IllegalAccessException
    { Class.forName("COM.ibm.db2.jdbc.app.DB2Driver").newInstance();
      con = DriverManager.getConnection("jdbc:db2:sample", "stadel", "stadel");
      stmt = con.createStatement();
    }
    public void insert() throws SQLException
    { stmt.executeUpdate("INSERT INTO ORG " +
        "VALUES(11, 'Head Office', 111, 'Corporate', 'San Jose')");
    }
    public void update() throws SQLException
    { stmt.executeUpdate("UPDATE ORG SET DEPTNAME = 'ITSO' WHERE DEPTNUMB=11");
    }
    public void delete() throws SQLException
    { stmt.executeUpdate("DELETE FROM ORG WHERE DEPTNUMB = 11");
    }
    public void orgList() throws SQLException
    { ResultSet rs = stmt.executeQuery("SELECT * FROM ORG");
      System.out.println("Organization List:");
      while (rs.next()) {
          int f1 = rs.getInt("DEPTNUMB");
          float f2 = rs.getFloat("MANAGER");
          String s1 = rs.getString("DEPTNAME");
          String s2 = rs.getString("DIVISION");
          String s3 = rs.getString("LOCATION");
          System.out.println(f1+"-"+s1+"-"+f2+"-"+s2+"-"+s3);
      }
    }
}

```

Figure 40. JDBC Insert, Update, and Delete Program

The result of the program is listed below. The highlighted line in the program's output indicates that a line has been inserted or updated.

```

--- ORG List Before Modifications ---
Organization List:
10-Head Office-160.0-Corporate-New York
15-New England-50.0-Eastern-Boston
20-Mid Atlantic-10.0-Eastern-Washington
38-South Atlantic-30.0-Eastern-Atlanta
42-Great Lakes-100.0-Midwest-Chicago
51-Plains-140.0-Midwest-Dallas
66-Pacific-270.0-Western-San Francisco
84-Mountain-290.0-Western-Denver

--- SQL Add DEPTNUMB 11 ---
Organization List:
10-Head Office-160.0-Corporate-New York
15-New England-50.0-Eastern-Boston
20-Mid Atlantic-10.0-Eastern-Washington
38-South Atlantic-30.0-Eastern-Atlanta
42-Great Lakes-100.0-Midwest-Chicago
51-Plains-140.0-Midwest-Dallas
66-Pacific-270.0-Western-San Francisco
84-Mountain-290.0-Western-Denver
11-Head Office-111.0-Corporate-San Jose

--- SQL Update DEPTNAME ---
Organization List:
10-Head Office-160.0-Corporate-New York
15-New England-50.0-Eastern-Boston
20-Mid Atlantic-10.0-Eastern-Washington
38-South Atlantic-30.0-Eastern-Atlanta
42-Great Lakes-100.0-Midwest-Chicago
51-Plains-140.0-Midwest-Dallas
66-Pacific-270.0-Western-San Francisco
84-Mountain-290.0-Western-Denver
11-ITS0-111.0-Corporate-San Jose

--- SQL Delete DEPTNUMB 11 ---
Organization List:
10-Head Office-160.0-Corporate-New York
15-New England-50.0-Eastern-Boston
20-Mid Atlantic-10.0-Eastern-Washington
38-South Atlantic-30.0-Eastern-Atlanta
42-Great Lakes-100.0-Midwest-Chicago
51-Plains-140.0-Midwest-Dallas
66-Pacific-270.0-Western-San Francisco
84-Mountain-290.0-Western-Denver

```

Statement and Prepared Statement

Now let us compare the Statement class with the PreparedStatement class. A PreparedStatement object is a precompiled SQL statement. Such an object can be used to efficiently execute the same statement multiple times, with different values of host variables.

Figure 41 shows a program written for both a Statement and PreparedStatement class. The result should be the same, but the PreparedStatement works more efficiently if you have to execute it often.

```
import java.sql.*;
public class StatementTest
{
    public static void main(String[] args)
    {
        StatementTest s;
        String drv = "COM.ibm.db2.jdbc.app.DB2Driver";
        String URL = "jdbc:db2:sample";
        try {
            s = new StatementTest();
            Class.forName(drv).newInstance();
            Connection con = DriverManager.getConnection(URL, "stade3", "stade3");

            Statement s1 = con.createStatement();
            ResultSet r1 = s1.executeQuery("SELECT * FROM ORG WHERE DEPTNUMB > 40");
            s.orgList("\n--- Org list using Statement ---", r1);

            PreparedStatement s2 = con.prepareStatement(
                "SELECT * FROM ORG WHERE DEPTNUMB > ?");
            s2.setInt(1,40); // set first host variable value
            ResultSet r2 = s2.executeQuery();
            s.orgList("\n--- Org List using PreparedStatement --- 40", r2);

            s2.setInt(1,60); // set host variable to 60
            ResultSet r3 = s2.executeQuery();
            s.orgList("\n--- Org List using PreparedStatement --- 60", r3);
        }
        catch(Exception e) { System.out.println("Exception: " + e); }
    }
    public void orgList(String s, ResultSet rs) throws SQLException
    {
        System.out.println(s);
        while (rs.next()) {
            int f1 = rs.getInt("DEPTNUMB");
            int f2 = rs.getInt("MANAGER");
            String s1 = rs.getString("DEPTNAME");
            String s2 = rs.getString("DIVISION");
            String s3 = rs.getString("LOCATION");
            System.out.println(f1+s1+f2+s2+s3);
        }
    }
}
```

Figure 41. Statement and PreparedStatement Classes

To create a `PreparedStatement` use an SQL statement with host variables denoted by a question mark (?):

```
PreparedStatement s2
    = con.prepareStatement("SELECT * FROM ORG WHERE DEPTNUMB > ?");
```

Before executing the statement, you must assign a value of matching type to each host variable:

```
s2.setInt(1,40);
```

The first parameter is the number of the host variable, the second, the value.

Notice that the `PreparedStatement` is executed twice in the program, each time with a different value of the host variable.

The results of the program are listed below. As you can see, the results are the same.

```
--- Org list using Statement ---
42-Great Lakes-100-Midwest-Chicago
51-Plains-140-Midwest-Dallas
66-Pacific-270-OWestern-San Francisco
84-Mountain-290-Western-Denver

--- Org List using PreparedStatement --- 40
42-Great Lakes-100-Midwest-Chicago
51-Plains-140-Midwest-Dallas
66-Pacific-270-OWestern-San Francisco
84-Mountain-290-Western-Denver

--- Org List using PreparedStatement --- 60
66-Pacific-270-OWestern-San Francisco
84-Mountain-290-Western-Denver
```

Callable Statement

Callable statements are used to interface with stored procedures. In many cases a stored procedure is a precompiled program with embedded static SQL statements. A stored procedure provides better performance than dynamic SQL written in Java with JDBC.

Some DBMSs also allow stored procedures written in Java. DB2 Universal Database (UDB) provides a sample Java program that calls a stored procedure written in Java. The description that follows is based on the `DB2Stp.java` sample program provided in `d:\SQLLIB\samples\java`.

A callable statement is created by using an SQL statement with parameters denoted by a question mark (?):

```
CallableStatement cstmt = con.prepareCall("CALL procname (?, ?, ?)");
```

Parameters of a stored procedure are defined in the DBMS catalog as in, out, or inout. Input parameters (in, inout) must be set, and output parameters (out, inout) must be registered before execution:

```
cstmt.setInt(1,40);
cstmt.setDouble(2, 4.58)
cstmt.registerOutParameter(2, java.sql.Types.DOUBLE);
cstmt.registerOutParameter(3, java.sql.Types.CHAR);
```

The first value in these calls is the number of the parameter in the stored procedure. The second is the value of an input parameter or the type of an output parameter.

A callable statement is a subclass of a prepared statement. To execute the stored procedure, we call one of the execution methods, `execute` or `executeQuery`:

```
cstmt.execute();
```

The results of output parameters are retrieved by parameter number:

```
double result2 = cstmt.getDouble(2);
String result3 = cstmt.getString(3);
```

A stored procedure can return one or multiple result sets. To retrieve a single result set we use `executeQuery`:

```
ResultSet rs = cstmt.executeQuery();
```

To retrieve multiple results sets we use the `getResultSet` and `getMoreResults` methods:

```
cstmt.execute();
ResultSet rs = cstmt.getResultSet();
// process result set ...
while ( getMoreResults() );
{   rs = cstmt.getResultSet();
    ... }
```

This short overview of callable statements is in no way complete but should give you a “feeling” for how stored procedures are called from a Java program.

JDBC in VisualAge for Java Enterprise

Up to now we have built a JDBC application and an applet manually, without using the sophisticated environment of VisualAge for Java Enterprise. In this environment, we can build a complete JDBC application or applet without writing any Java JDBC code.

In “JDBC Applet” on page 50, we intentionally built a simple JDBC applet, without using the sophistication of the GUI system provided by the Java Abstract Windowing Toolkit (AWT) package.

Now, with VisualAge for Java Enterprise in our pocket, we can embark on a more ambitious project. The Data Access Builder generates JDBC-enabled Java beans for us that will make the composition of applications and applets very easy.

In Chapter 4, “Data Access Builder,” on page 59, we explain step-by-step how you can build the organization applet in Figure 39 on page 51 and use the Data Access Builder to map classes from a database table.

4

Data Access Builder

In this chapter we provide detailed information about Data Access Builder, the VisualAge for Java tool that enables Java applications or applets to access and retrieve data from an SQL database.

We start by showing how easy it is to use Data Access Builder. We explain what Data Access Builder does for you and what the logic behind the generated classes is. In this way, you can appreciate the power and the usefulness of this tool.

Relational Database Access

The Data Access Builder is an application development tool that you can use to create data access classes customized for your existing relational database tables. In other words, Data Access Builder generates Java beans that access data outside the Java environment. Do not modify the source code that Data Access Builder creates; just use the generated classes in your programs.

Data Access Builder generates code that uses JDBC to access your database. You can use the DB2 JDBC driver, the JDBC-ODBC Bridge of JDK Version 1.1 or later, or other JDBC drivers with the generated code.

Data Access Builder generates code from database tables, database views, or any valid SQL statements that you enter. For example, you can simply specify a database table name, and Data Access Builder accesses the table information and generates Java source code that enables you to add, update, delete, or retrieve the data in that table.

You can tailor how Data Access Builder builds the code before code generation. You can, for example, remove unused columns or add your own methods to the beans to be generated. If you want to use a more advanced mapping schema than table-to-class, you can specify your own SQL statements, for example, join operations.

The generated classes enable you to use a cursor to fetch rows from database queries that return result sets. Separate classes provide services for connecting to and disconnecting from your databases. In addition, commit and rollback methods are generated to handle transaction services.

Before starting, review “Prerequisites for JDBC Applications” on page 360.

Building a JDBC Application

Now we guide you in building the JDBC applet that we showed in Figure 39 on page 51, using Data Access Builder.

Application Requirements

Let us construct a JDBC applet in which, at the click of a push button, all rows from the organization table are retrieved and displayed in a multicolumn list box.

Remember, you can always divide any application into three parts or modules: the user interface, the business logic, and the data store.

For simplicity, in our case we combine the data store and the business logic into one module and put the user interface in another module. Figure 42 depicts the structure of the JDBC applet that we are going to build.

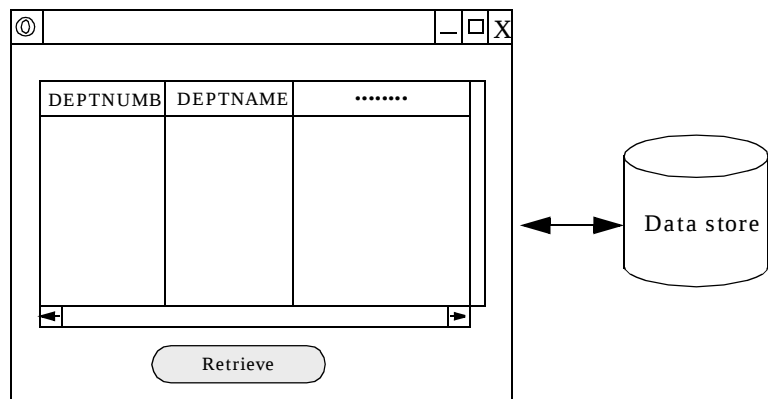


Figure 42. Simple JDBC Applet GUI Design

Now, to make everything pure object-oriented, we need to wrap the ORG table into Java beans. In VisualAge for Java Enterprise, we can do this by mapping the schema of the ORG table, using Data Access Builder.

After creating a class mapping for the ORG table, we construct the user interface for the applet. The interaction between the user interface and the database classes enables us to retrieve the rows of the ORG table and to display the result (Figure 43).

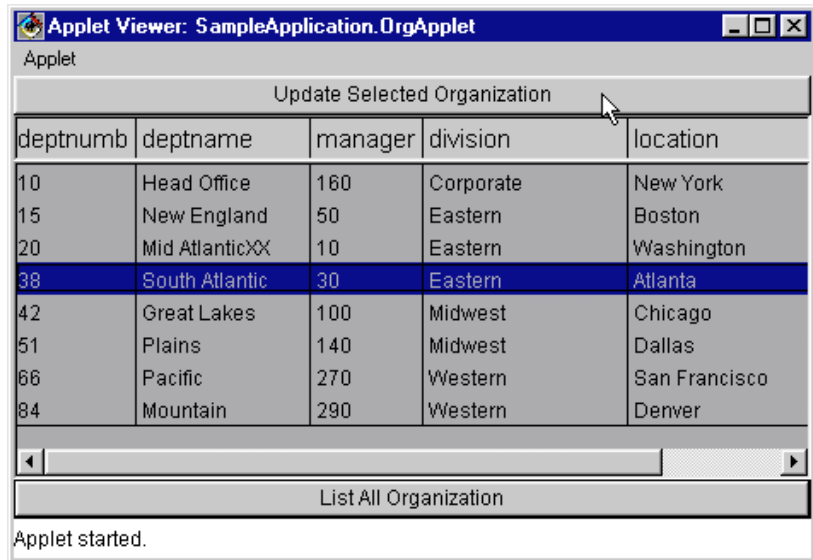


Figure 43. The JDBC Applet Result

We also show you how to incorporate ready-to-use GUI components that will run the basic SQL functions, such as add, delete, retrieve, and update of your generated beans. We use these GUI components in the Visual Composition Editor of VisualAge for Java in our own applications (Figure 44).

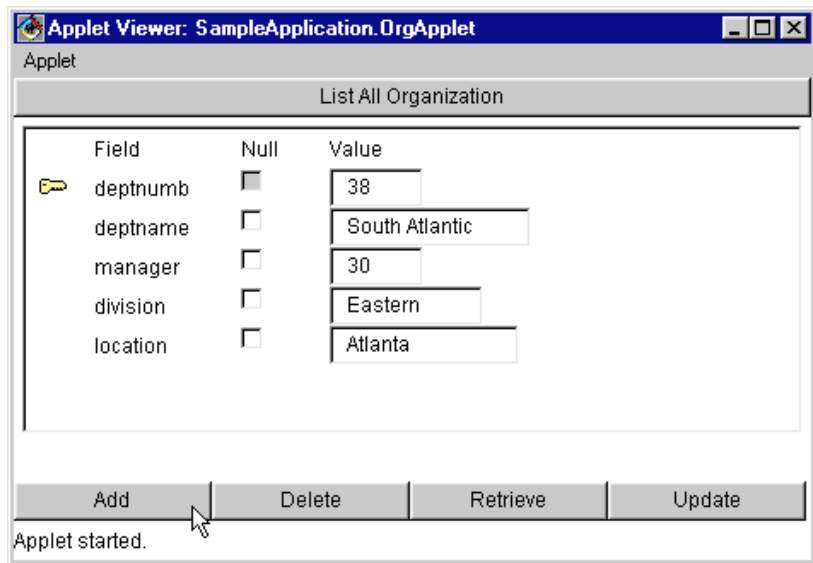


Figure 44. Application with Ready-to-Use GUI Components

Development Process with Data Access Builder

Follow these high-level steps to construct an application (or applet) with Data Access Builder (see Figure 45):

1. Create a project (or use an existing one) in the Workbench.
2. Create a package under the project. We suggest putting the data access beans into a project separate from the application.
3. Start Data Access Builder in the package.
4. Create a mapping, using table definitions from the relational catalog (1) and SQL statements (2).
5. Generate the data access beans from the mapping (3).
6. Create the application's GUI (4).
7. Use the data access beans to complete the application (5).

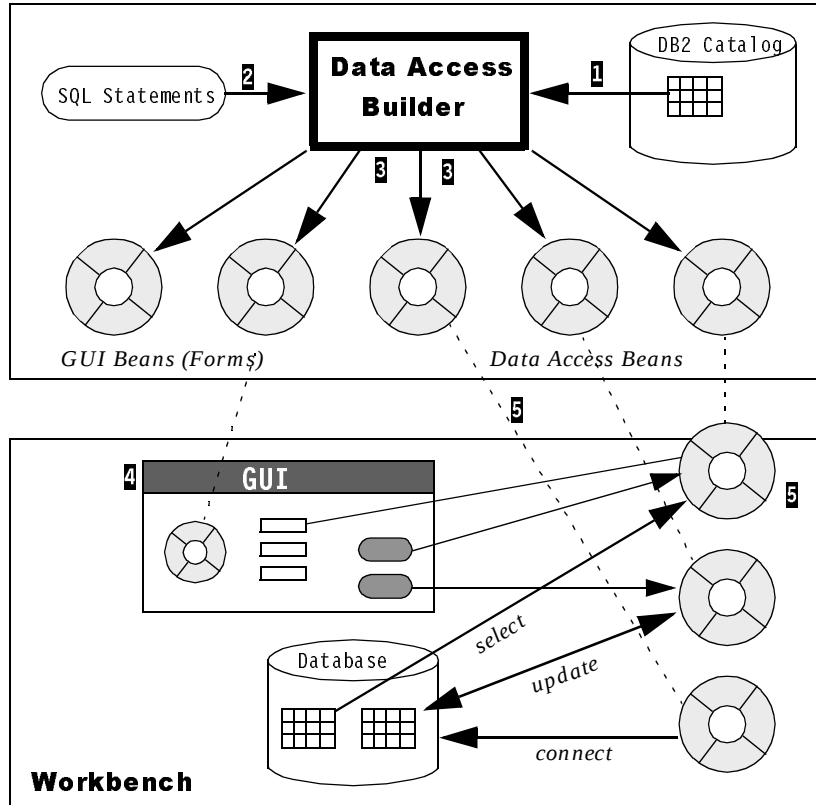


Figure 45. Development Process with Data Access Builder

Using Data Access Builder for the Organization Applet

Let us use the development process with Data Access Builder for our organization applet:

1. Create a project in the workbench, for example, JDBC.
2. Create two packages in the JDBC project, for example, SampleApplication and SampleDax.
3. Start Data Access Builder for the SampleDax package.
4. Map the ORG table into data access beans
5. Generate the data access beans.
6. Analyze and understand the generated data access beans.
7. Create the application or applet, using some of the generated data access beans.

If you do not know how to perform steps 1 and 2, refer to the IDE HTML Help, which comes with VisualAge for Java.

Before you continue with step 4, be sure to create the JDBC project and the two packages in the workbench, as shown in Figure 46.

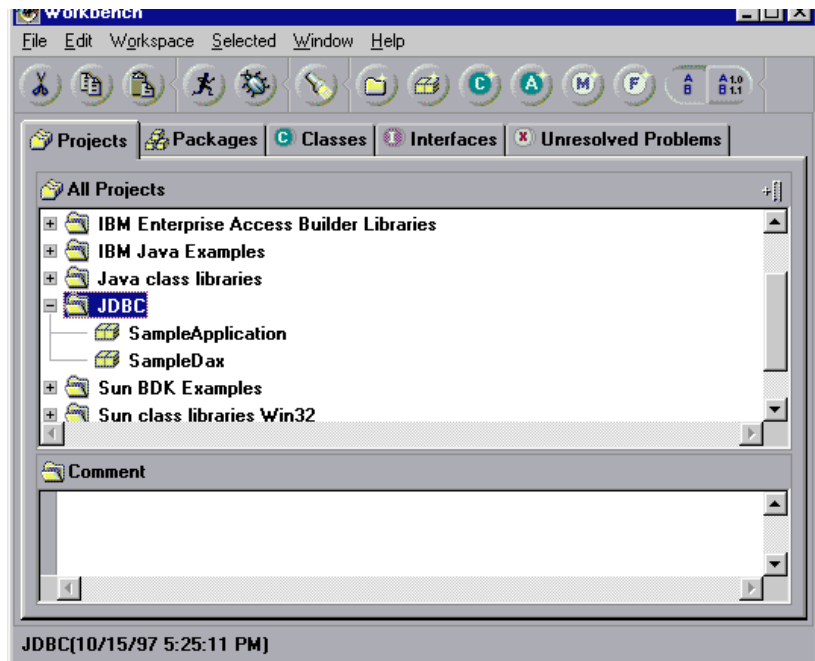


Figure 46. JDBC Project in the Workbench

Starting Data Access Builder

To start Data Access Builder and to create data access beans, select the SampleDax package in the Workbench, and select *Tools -> Data Access -> Create Data Access Beans...* in the *Selected* menu (Figure 47).

The generated beans have to reside in one package. That is why you do not see this menu option if no package is selected.

Alternatively you can use the context (pop-up) menu of the selected package.

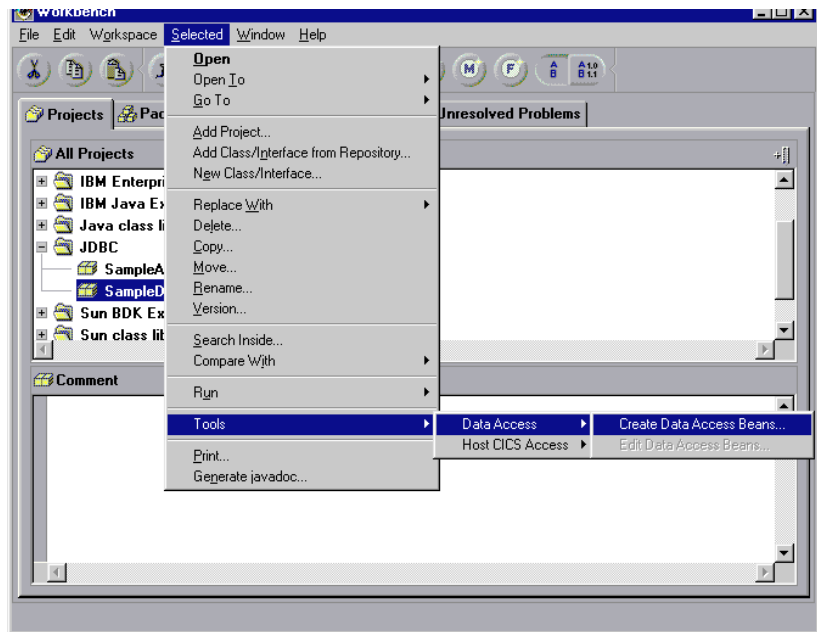


Figure 47. Launching Data Access Builder

Selecting the *Create Data Access Beans...* option, launches Data Access Builder, which you can use to create a new Data Access Builder schema mapping.

The schema mapping is visually represented in the Data Access Builder window (Figure 48) by icons. At this time only the package icon is shown.

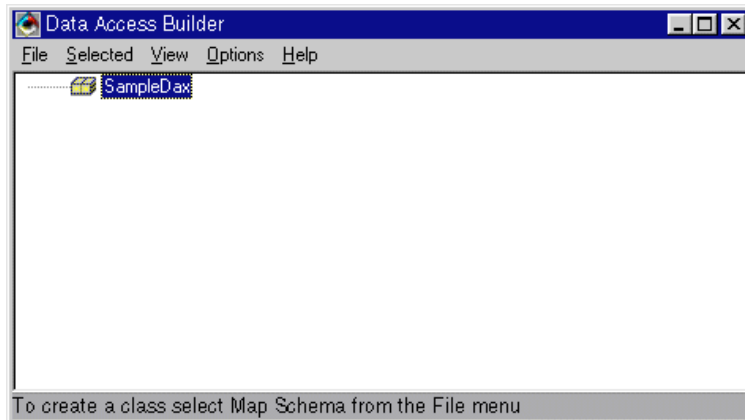


Figure 48. Data Access Builder

Mapping a Table into Data Access Beans

To create a schema mapping, select *Map Schema...* in the *Selected* menu of the Data Access Builder window or use the pop-up menu.

Click on **Next** in the Data Access Builder SmartGuide window to go to the Select database and mapping method window (Figure 49). Select *DB2* as the database source, and *SAMPLE* as the database. Leave the default value of the preselected radio button.

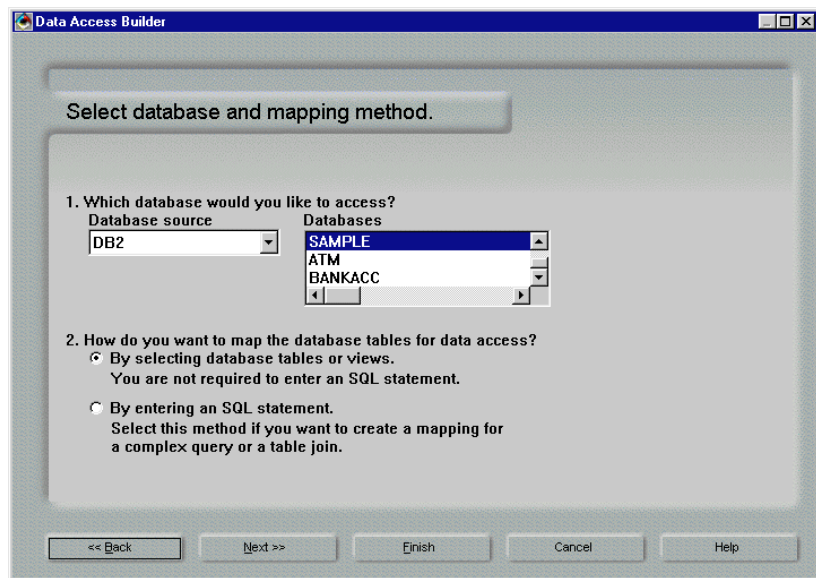


Figure 49. Database and Mapping Method Selection

Note that relational database systems other than DB2 are supported. One of the drop-down choices is ODBC.

Click on **Next** in the Select database and mapping method window to go to the Select the table(s) for mapping window (Figure 50). For Get tables using the specified filter, type in the owner name (creator of the DB2 sample database, STADE3 in our case), table name, and table type, and click on the **Get tables** button (left of the Filter fields). Select *ORG* as the table to be accessed.

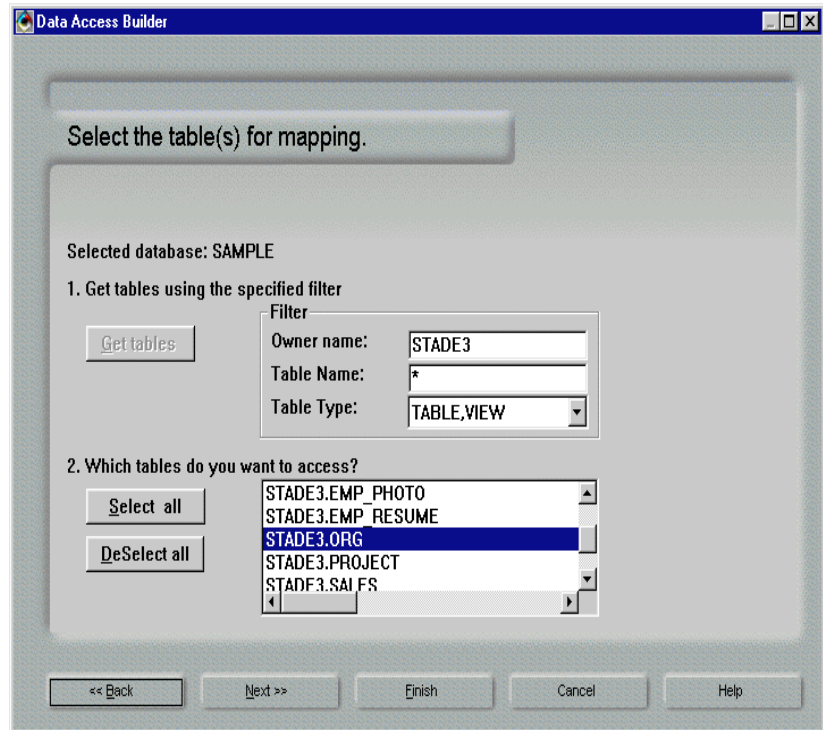


Figure 50. Table Mapping Selection

You can click on the **Next** button in the Select the table(s) for mapping window to go to the Schema Mapping Description window, where you can describe your schema mapping with text. The description will be added as a comment in the generated classes. Click on the **Finish** button when you are ready.

The SmartGuide tool brings you back to the Data Access Builder window and creates the necessary mapping as specified during the session. Each mapping is represented by an icon in a tree structure (Figure 51). If you do not see the Org bean's attributes, click on the plus sign (+) next to the Org bean.

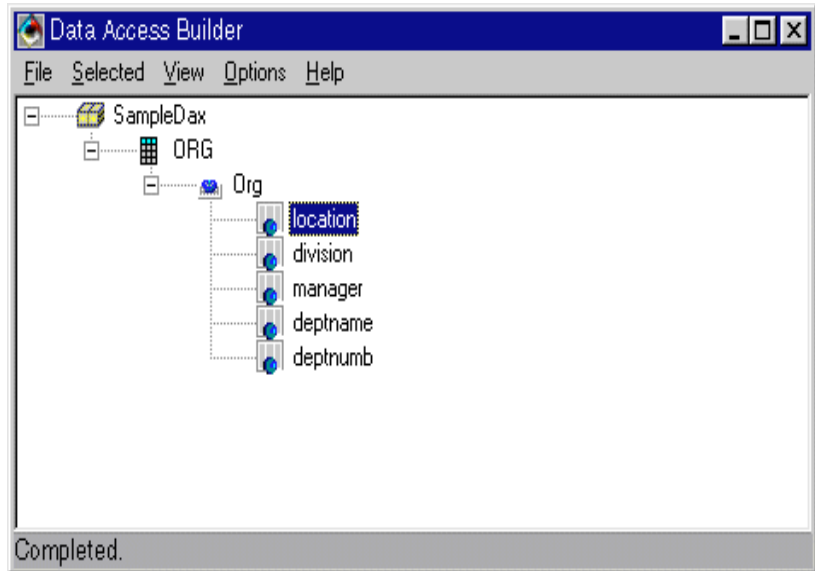


Figure 51. Generated Org Bean and Attributes

The Data Access Builder does not generate delete, update, or retrieve methods for a table without a key, that is, a table without a column (or group of columns) that uniquely identifies each row. Usually, Data Access Builder maps a table's primary key if one is defined in the DB2 catalog.

Unfortunately, as indicated in Figure 51, a primary key was not defined during definition of the ORG table. Otherwise, at least one icon would symbolize the presence of a unique key (see Figure 53 on page 69). Therefore we need some way of forcing the Data Access Builder to generate the methods we require.

Fortunately, the Data Access Builder provides an easy way of specifying a key. We can modify one or more attributes to form the key. We select the `deptnumb` attribute as a unique key for our application. We click with the right mouse button on the `deptnumb` attribute and select *properties*. In the Properties window (Figure 52), we check the *Data Identifier* checkbox to indicated our desire and click on **OK**.

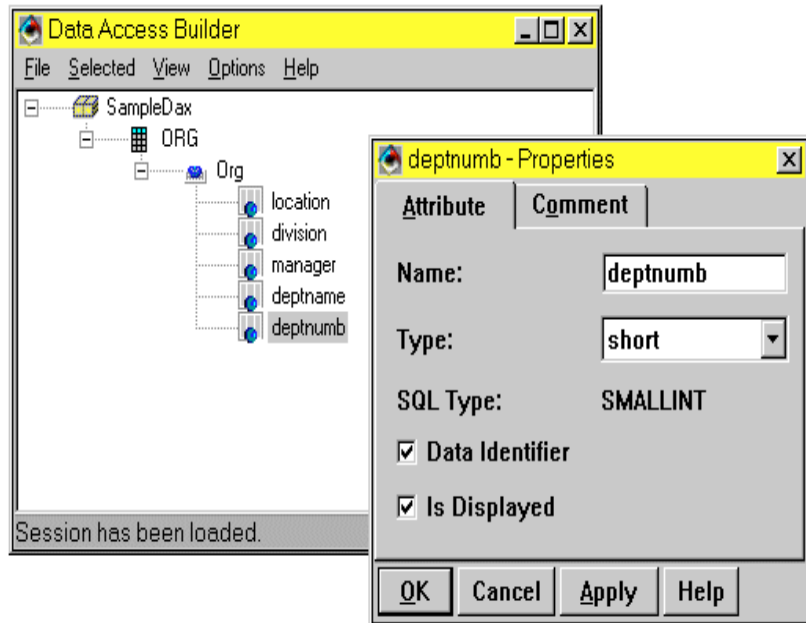


Figure 52. Specifying a Unique Data Identifier

After we modified the deptnumb attribute to form the data identifier, Data Access Builder remaps and redraws the deptnumb icon. Now it shows a notebook with tabs to indicate that the deptnumb attribute is a unique key (Figure 53).

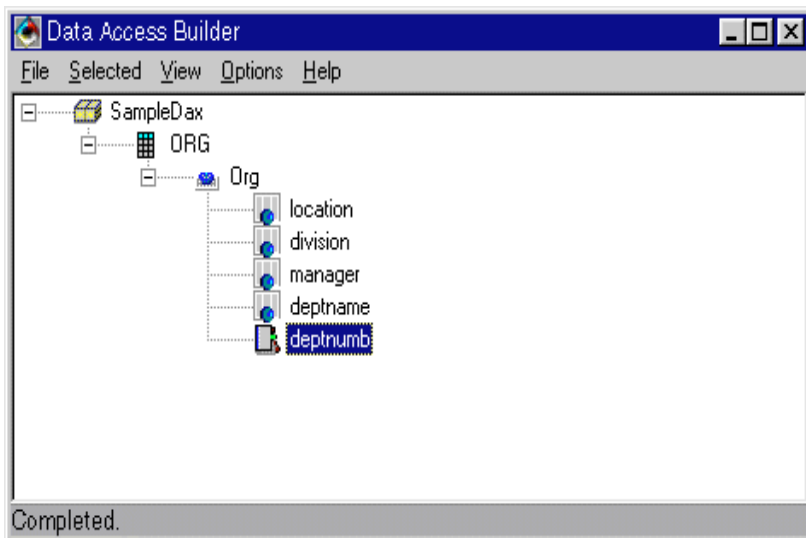


Figure 53. Org Bean with Unique Data Identifier

Before we ask Data Access Builder to generate the classes, there are two other important items we need to specify: the JDBC driver we intend to use, and the URL where the database resides. Although you could specify these two items at run time (see Figure 58 on page 76), it is good practice to initialize them.

In our sample, we use the following values:

- ❑ For the driver, `COM.ibm.db2.jdbc.net.DB2Driver`
- ❑ For the database URL, `jdbc:db2://localhost:8888/SAMPLE`

To specify these values using the right mouse button, click on the Org bean icon and select *Properties*. In the Properties window (Figure 54), select the **Access** tab and under Connection Information, specify the driver and the URL. You may want to remove the table qualifier in the **Source** tab, so that your application is independent of the table qualifier.

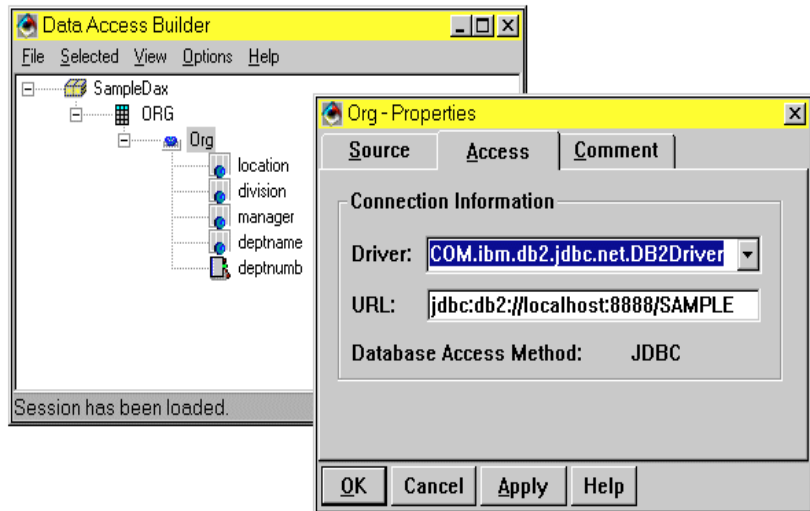


Figure 54. Connection Information in the Org Bean Properties Window

Generating the Data Access Beans

After specifying all of the required information for the Org bean, you can generate the Java classes. Select *File* in the tool bar of the Data Access Builder window and then select *Save and Generate*. Select *File -> Exit* when Data Access Builder has finished generating the classes.

The Java source code that Data Access Builder generates is automatically imported and compiled in the package in the Workbench.

For this example, Data Access Builder generates 17 classes that encapsulate the database access for our application. We do not have to write any SQL code in the application; it interacts with the database through the beans and classes generated by Data Access Builder (Figure 55).

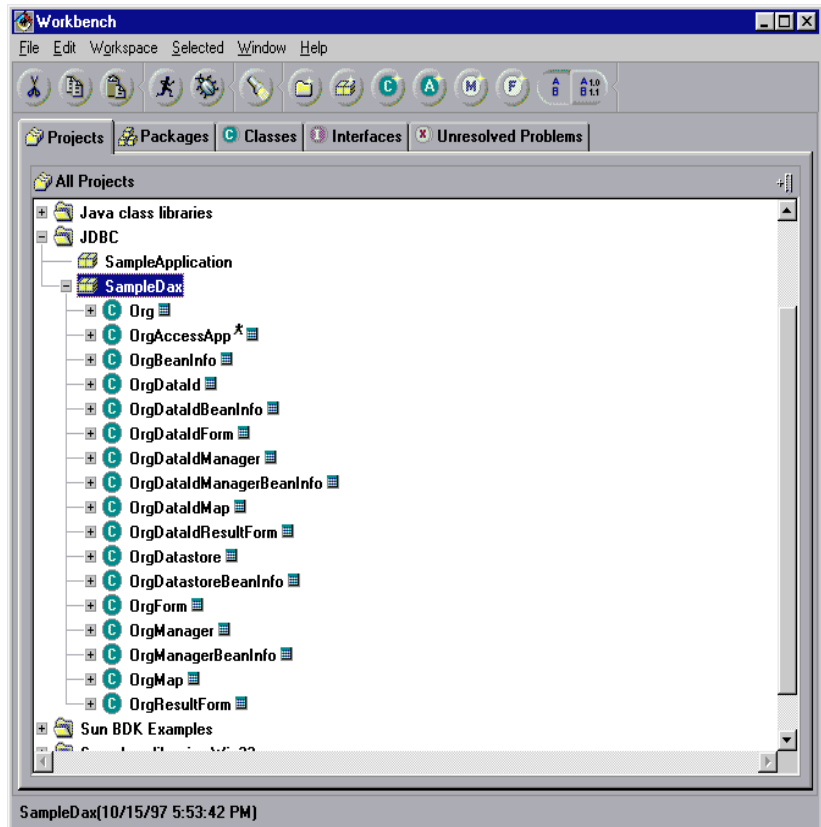


Figure 55. Beans and Classes Generated by Data Access Builder

Data Access Builder Beans and Classes

Table 9 lists the beans generated by Data Access Builder. The `<class>` template stands for the name of the mapping given in the Data Access Builder.

Table 9. Beans Generated by Data Access Builder

Class Name	Extends	Description
<code><class></code> <code>Org</code>	<code>PersistentObject</code>	Objects represent one row from the schema. Class contains database access methods, including any user-defined methods you have defined.
<code><class>Manager</code> <code>OrgManager</code>	<code>DAManager</code>	Enables you to select a collection of rows from the tables and work with them, or to open a database cursor and access a collection of rows one at a time. Class contains any user-defined manager methods you have defined.
<code><class>DataId</code> <code>OrgDataId</code>	<code>PODataId</code>	Objects represent the set of columns that uniquely identify a row. This class is generated only if the mapping specifies a data ID column or columns.
<code><class>DataIdManager</code> <code>OrgDataIdManager</code>	<code>DAManager</code>	Enables you to select a collection of data IDs from the table and work with them, or to open a database cursor and access a collection of data IDs one at a time. This class is generated only if the mapping specifies a data ID column or columns.
<code><class>Datastore</code> <code>OrgDatastore</code>	<code>DatastoreJDBC</code>	Objects represent connections to a database. Class contains methods for connect, disconnect, commit, and rollback operations.

In addition to the beans (classes) described in Table 9, there are three interesting types of classes generated by Data Access Builder:

- ☐ Data access BeanInfo classes
- ☐ Form classes
- ☐ Access application class

BeanInfo Classes

BeanInfo classes enable builders, such as the Visual Composition Editor, to use the data access classes by providing information about the properties, methods, and events of each class. These BeanInfo classes let the builder know what can be done with the corresponding class of the bean. Data Access Builder generates a BeanInfo class for all the classes described in Table 9:

- ❑ <class>BeanInfo: OrgBeanInfo
- ❑ <class>ManagerBeanInfo: OrgManagerBeanInfo
- ❑ <class>DataIdBeanInfo: OrgDataIdBeanInfo
- ❑ <class>DataIdManagerBeanInfo: OrgDataIdManagerBeanInfo
- ❑ <class>DatastoreBeanInfo: OrgDataStoreBeanInfo

When you create a new bean, VisualAge for Java automatically generates a BeanInfo class for it. All of these BeanInfo classes follow the Java beans standard.

Form Classes

Data Access Builder generates a group of Form classes that are GUI parts you can use in your own applets and applications (see Table 10).

Table 10. Form Classes

Class Name	Description
<class>Form OrgForm	A form for displaying the attributes and attribute values of a <class> object (which corresponds to a row in the database table); lets you change the attribute values, add, update, delete, retrieve, and fetch data access objects.
<class>DataIdForm OrgDataIdForm	A form for displaying the attributes and attribute values of a <class>DataId object; lets you change the attribute values, add, update, delete, retrieve, and fetch data access objects.
<class>ResultForm OrgResultForm	A form for displaying a collection of created <class> objects that were retrieved with a select or open method; lets you choose one object(row) and display it in the <class>Form.
<class>DataIdResultForm OrgDataIdResultForm	A form for displaying a collection of <class>DataId objects retrieved with a select or open method; let you choose one object and display it in a <class>Form or a <class>DataIdForm.

For your convenience, Figure 56 shows all the forms generated by Data Access Builder for our sample.

`<class>DataIdForm`

Field	Null	Value
deptnumb	☐	0

`<class>DataIdResultForm`

deptnumb	
----------	--

`<class>Form`

Field	Null	Value
deptnumb	☐	0
deptname	☒	
manager	☒	
division	☒	
location	☒	

`<class>ResultForm`

deptnumb	deptname	manager	division	location	
----------	----------	---------	----------	----------	--

Figure 56. Forms Generated by Data Access Builder

There is one additional form not shown in Figure 56—the `IConnectPanel`, which is in the `COM.ibm.ivj.eab.data` package of the IBM Enterprise Access Builder Library in your Workbench.

The connection panel is similar to the Form classes. You can use it in an application to enter database access information, such as the URL, driver, user ID, and password. It lets you connect, disconnect, commit, roll back, and set the `autoCommit` setting (Figure 57).

The figure shows a 'Database Connection Panel' dialog box. It contains the following elements:

- URL:** jdbc:db2://localhost:8888/SAMPLE
- Driver:** COM.ibm.db2.jdbc.net.DB2Driver
- UserId:** USERID
- Password:** *****
- Autocommit:** A checked checkbox.
- Status:** Disconnected
- Buttons:** Connect, Disconnect, and Cancel.

Figure 57. Database Connection Panel

Access Application Class

With the information you provided during the mapping, Data Access Builder constructs a complete application against your table. The application is called `<class>AccessApp` (`OrgAccessApp` in our case).

Using the access application, you can connect, disconnect, commit, and roll back a database; manipulate a table (add, delete, update, and retrieve) in the connected database; add your own clause (called *suffix*) to the SQL select statement; open a cursor; step through the cursor for update or delete; list all rows retrieved, using the SQL suffix; and manipulate a selected row. Data Access Builder automatically generates all of these facilities for you. You also get the history of all of your activities, while you are connected to the database.

The access application uses a card layout that displays the individual pages (panels and forms) one at a time. A series of push buttons at the top enables you to navigate from page to page. The pages are:

- ❑ **Datastore**, for database operations, such as connect, disconnect, commit, and rollback (Figure 58)
- ❑ **Org**, for row manipulation, such as retrieve, insert, update, and delete (Figure 59)
- ❑ **Manager**, to execute a select statement with an optional WHERE clause (Figure 60)

- ❑ **Cursor**, to step through the rows retrieved in the Manager page, with update and delete of the fetched row (Figure 61)
- ❑ **ResultForm**, to display the rows retrieved in the Manager page in a tablelike format (Figure 62)
- ❑ **Selected**, to display a row selected in the ResultForm page and to step forward and backward through the rows retrieved in the Manager page (Figure 63)

Let us start with the Datastore page (Figure 58).

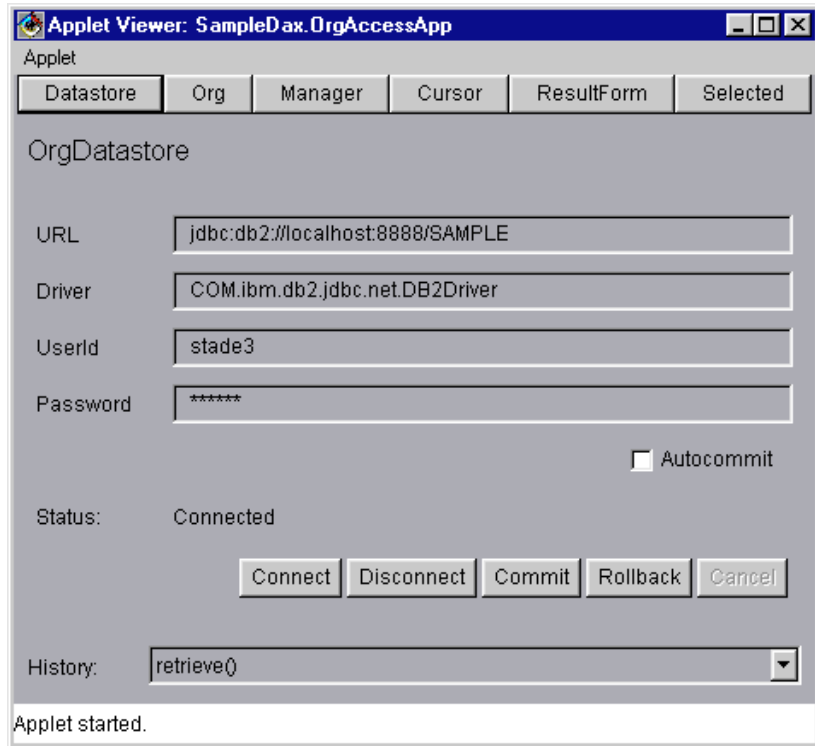


Figure 58. Access Application: Datastore Page

If you check the *Autocommit* check box, the **Rollback** and **Commit** buttons are removed. All transactions are automatically committed.

With the **Org** button selected, you can manipulate the table by executing add, retrieve, delete, and update functions. These functions are represented as a drop-down list labeled Action (Figure 59).

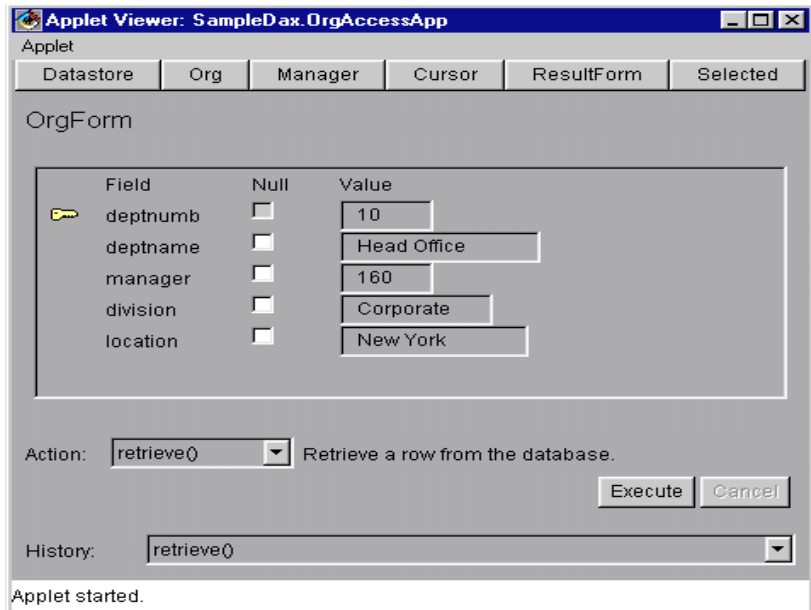


Figure 59. Access Application: Org (Row Manipulation)

With the **Manager** button selected, you can specify an SQL suffix (for example, a WHERE clause) for the select statement, execute the select, and open a cursor on the retrieved rows (Figure 60).

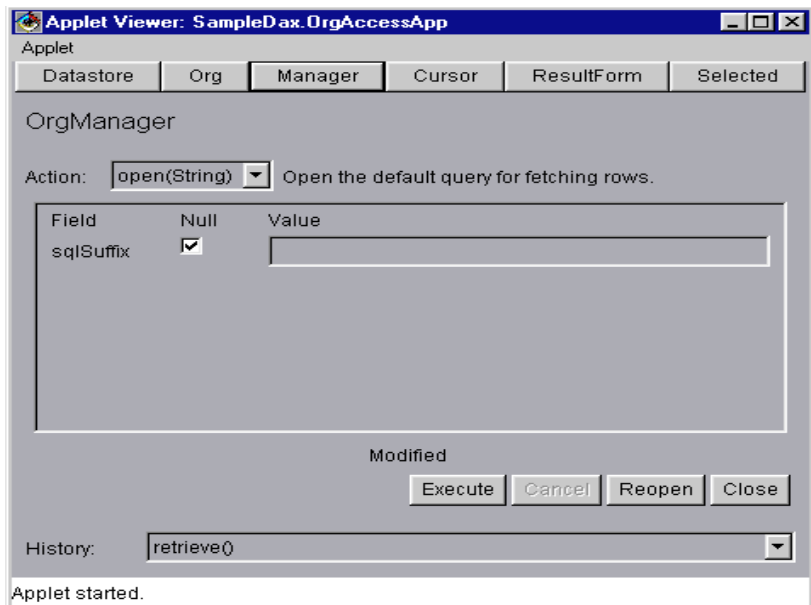


Figure 60. Access Application: Manager

With the **Cursor** button selected, you can update or delete a fetched row, and fetch the next row (Figure 61).

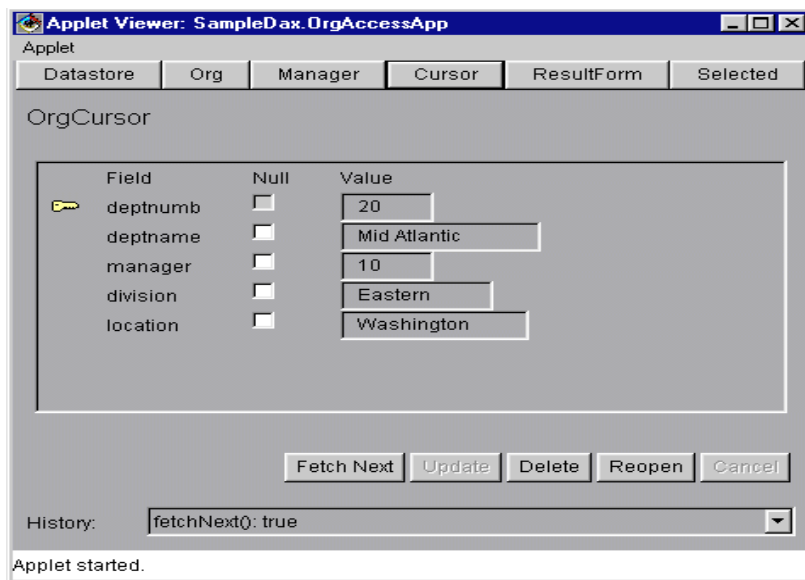


Figure 61. Access Application: Cursor

With the **ResultForm** button selected, you can fill the ResultForm with the cursor that you have opened in the Manager page, to display all the rows starting from the cursor (Figure 62).

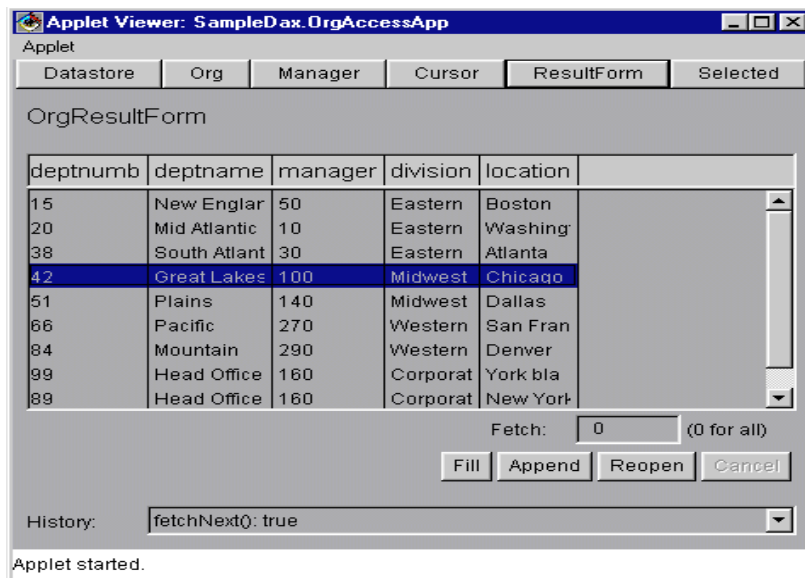


Figure 62. Access Application: ResultForm

You can select a row, and continue with the **Selected** page, where the row selected in the **ResultForm** is displayed. In this page, you can update and delete the row, step forward and backward through the rows displayed in the ResultForm, and you can perform the same row operations as on the **Org** page (Figure 63).

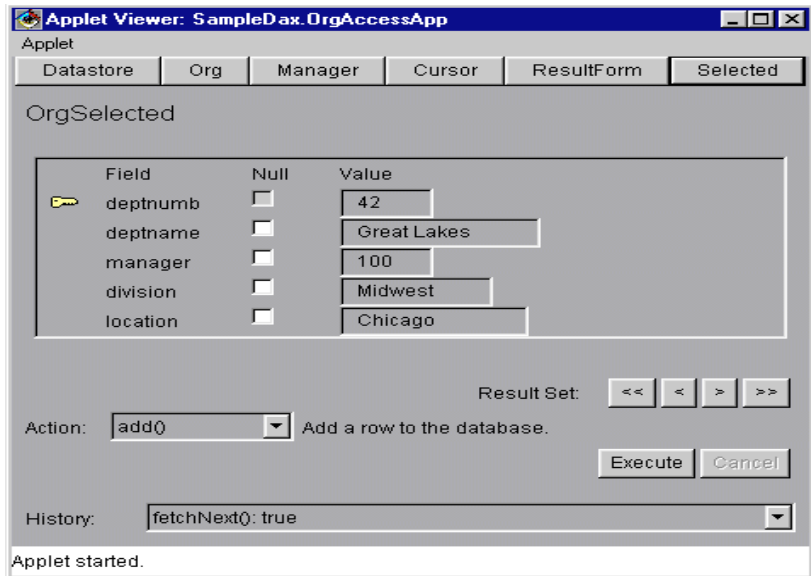


Figure 63. Access Application: Selected

In summary, the access application provides you with an easy way of testing the generated data access beans and the connection to the database.

The access application is not designed to be put into the hands of end users; they should not be concerned with JDBC drivers, database URLs, and connections. However, the access application and the generated forms can be of great help to you, as the developer of the real applications.

Creating the JDBC Sample Application and Applet

The best way to learn how to build a new application is to start with an existing application, because you know what the end result will be and you will never be misguided. Therefore in this section we “tear” apart the `<class>AccessApp` application and show you step by step how you can build an application similar to it. We use most of the parts provided by Data Access Builder.

Applet Overview

After you start the application as an applet, click the **List All Organization** push button to list all of the information from the Sample database table into the multicolumn list box (Figure 64).

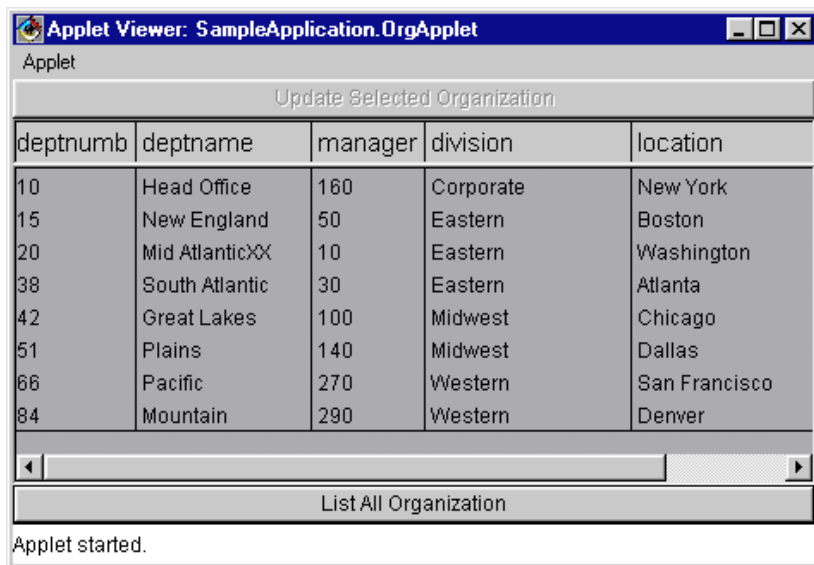


Figure 64. Organization Applet: Main Panel

Initially, no row is selected and the **Update Selected Organization** push button is disabled. When you select a row, for example, department number 38, the push button is enabled (Figure 65).

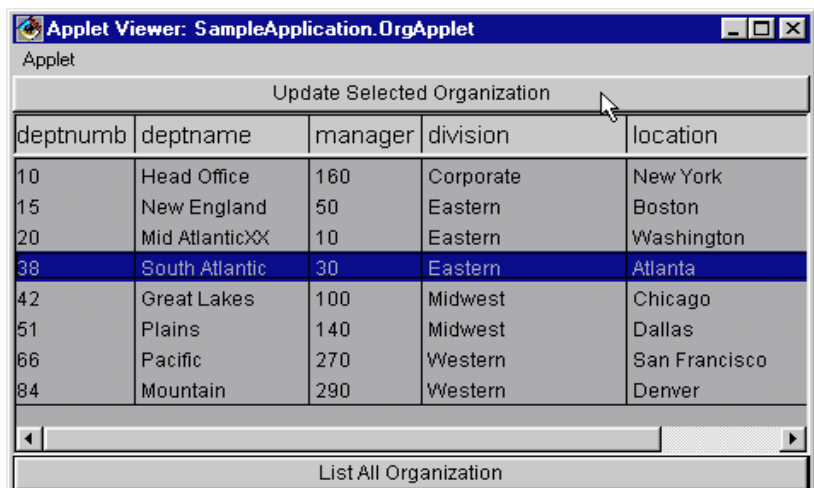


Figure 65. Organization Applet: Selecting a Row

When you click on the **Update Selected Organization** push button, you are presented with the second panel, which displays the row you selected. In this panel, you can add, delete, and update the row data. You can also retrieve another row by specifying a department number in the corresponding entry field (Figure 66).

When you click on the **List All Organization** push button, you return to the main panel and the list of departments is automatically refreshed (Figure 64).

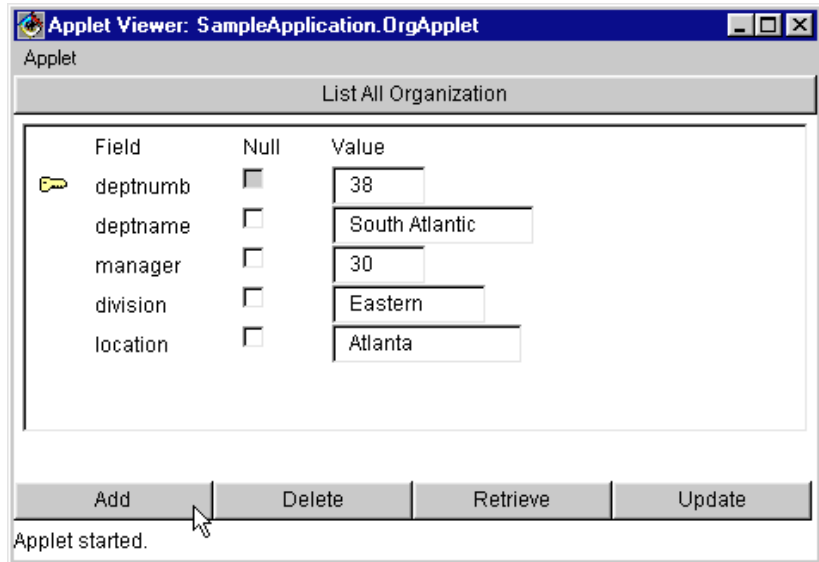


Figure 66. Organization Applet: Selected Row

Applet Construction

We construct the user interface of the applet as illustrated in Figure 67. From the Workbench, under the JDBC project, select the SampleApplication package (Figure 46 on page 64). Create a new applet and call it OrgApplet.

Change the layout of the default panel to CardLayout, and name the panel MainPanel. Drop two panels (from the Container category) into the default panel. Name the first PanelA, and the second, PanelB. Change the layout manager of both panels to BorderLayout.

If you experience difficulties accessing the individual panels, open the Beans List window (from the *Tools* pull-down, or using the icon button in the tool bar).

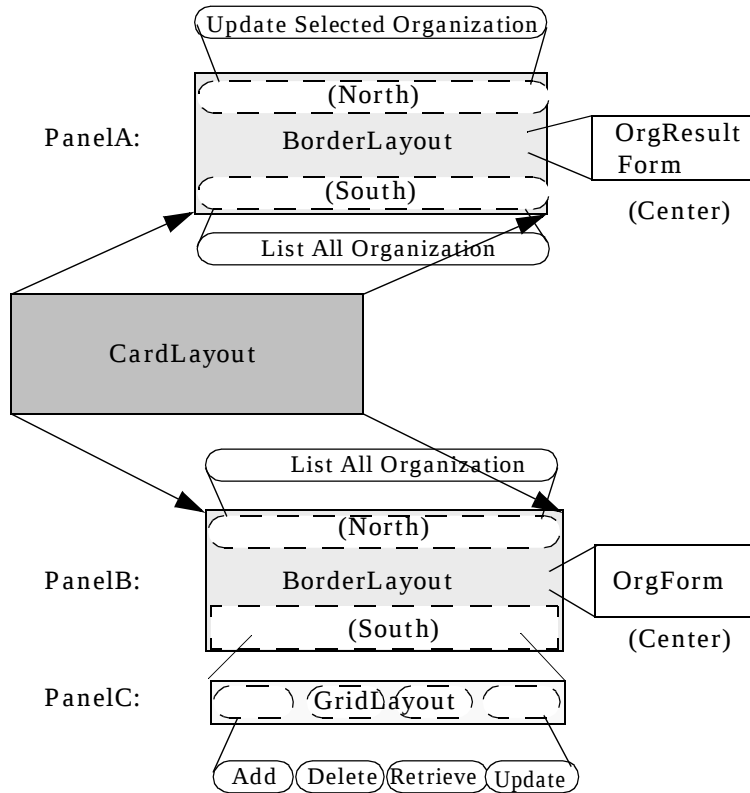


Figure 67. Organization Applet: User Interface Design

Add two push buttons in PanelA, one in the North area and the other in the South area. Label the North button with “Update Selected Organization” and name it UpdateSelectedButton. The South button has the label “List All Organization” and the name ListAllButton1.

Add one push button in the North of PanelB, label it “List All Organization,” and name it ListAllButton2. Add another panel (PanelC) on the free-form surface and change the layout to GridLayout. Add four push buttons to PanelC. Label the buttons “Add,” “Delete,” “Retrieve,” and “Update.” Now move PanelC, including the buttons, into the South area of PanelB.

To complete the user interface, add two visual beans from the SampleDax package; they were generated previously by Data Access Builder. The first bean is the OrgResultForm bean; add this bean to the center area of PanelA. The second bean is the OrgForm bean; add it to the center area of PanelB.

PanelA should now look like that in Figure 68, and PanelB should look like that in Figure 69.

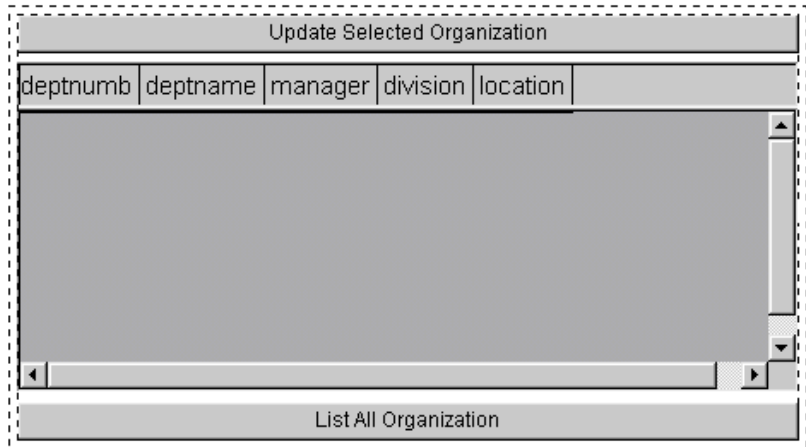


Figure 68. Organization Applet: Organization List Panel (PanelA)

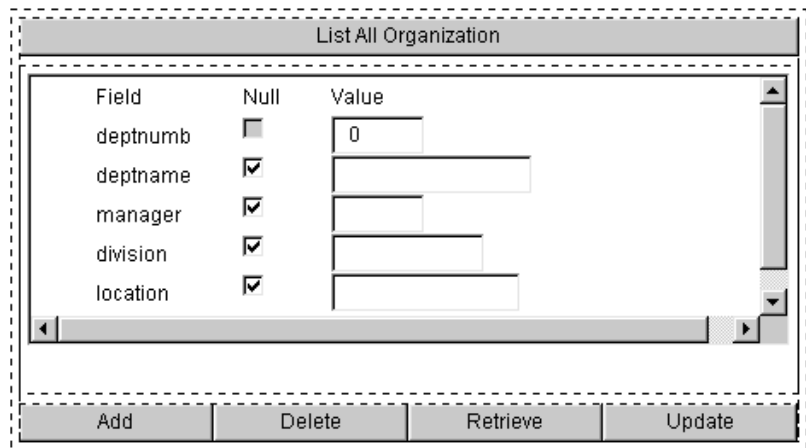


Figure 69. Organization Applet: Organization Detail Panel (PanelB)

To activate the **List All Organization** push buttons on both panels (PanelA and PanelB), you have to add a variable of type CardLayout to the free-form surface and connect it in the following way (see Figure 70):

- ❑ Connect the layout property of the MainPanel with the this property of the CardLayout variable (1).
- ❑ Connect the actionPerformed event of the **UpdateSelected-Button** (on PanelA) with the next method of the CardLayout variable (2). Pass the MainPanel as the parent parameter of the connection (3).

- ❑ Connect the **ListAllButton** (on PanelB) with the next method of the CardLayout variable. Pass the MainPanel as the parent parameter of the connection (same as 2 and 3, but for PanelB).

After you perform these three steps, the CardLayout connectivity should look like that in Figure 70.

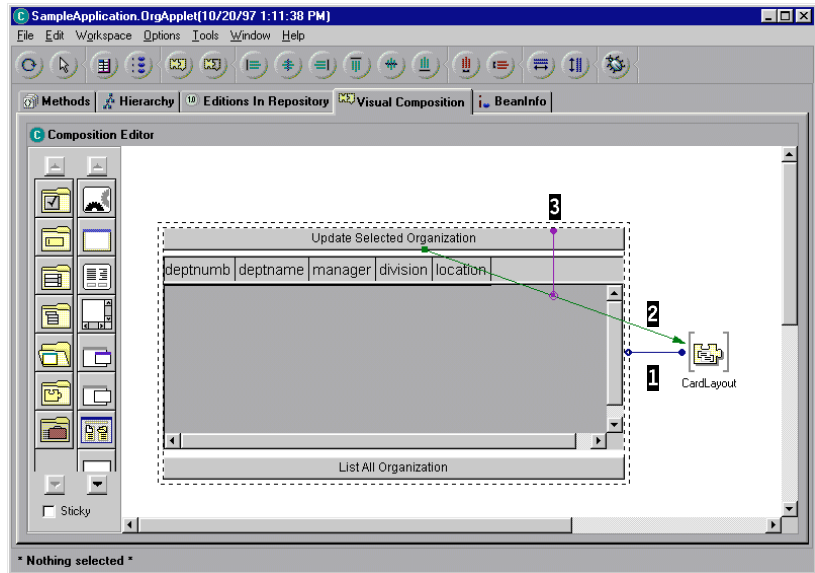


Figure 70. Organization Applet: CardLayout Connectivity

Note: In future screen captures we show only the *inside* of the Visual Composition Editor.

You can click on the **Test** icon button in the SampleApplication.OrgApplet window. When the application is started, click on the push button in the top pane, and the next panel is displayed.

Connecting to the Database

After you have tested the panel design of your applet, you can connect to the database.

Add an OrgDatastore bean from the SampleDax package to the free-form surface. To display potential error messages, add a message box bean from the palette (Enterprise Access category) to the free-form surface.

To create the logic for the database connection, make the following connections in the Visual Composition Editor (see Figure 71):

- ❑ Connect the `init` event of the applet with the `connect` method of the `OrgDatastore` bean (1). Note that the `init` event is an expert feature (click on the *Show expert features* check box). Set the parameters of the `connect` method to your user ID and password.
- ❑ Connect the `exceptionOccurred` event of the previous connection with the message box (2). Open the new connection and select the *Pass event data* check box.
- ❑ Connect the `destroy` event of the applet with the `disconnect` method of the `OrgDatastore` (3).

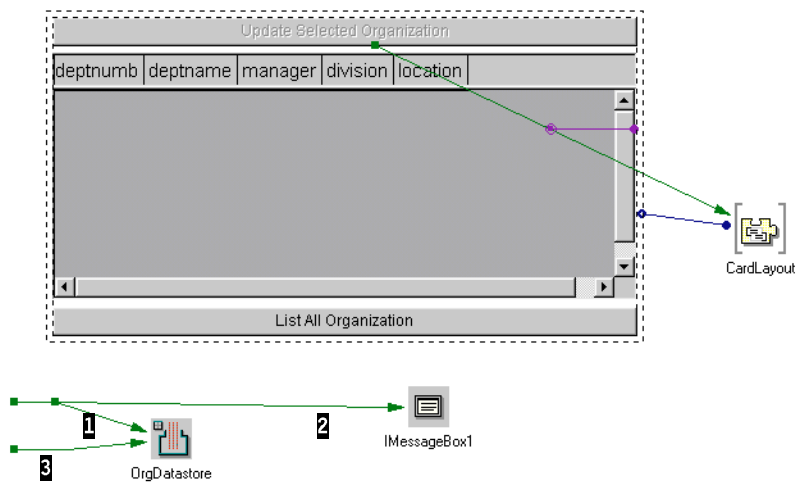


Figure 71. Organization Applet: Database Connection

Test the database connection of your applet. You should be able to connect to the database without prompting an error message box; otherwise check that DB2 and the JDBC daemon (DB2JSTRT 8888) are running and that you can connect to the database with the specified user ID and password.

Completing the Organization List Panel

As indicated in Figure 67 on page 82, the objective of the first panel (PanelA) is to enable the user to list all records in the multi-column list box, by clicking the **List All Organization** button. The user then selects a record and clicks the **Update Selected Organization** button, and the selected record is passed to the second panel (PanelB).

You need three beans to process PanelA:

- ❑ To populate the multicolumn list box with the organization's records, you have to add an OrgManager bean from the SampleDax package.
- ❑ To pass the selected record from PanelA to PanelB, you need a variable to hold the row object. This variable is of the type Org from the SampleDax package.
- ❑ When switching from PanelA to PanelB and vice versa, you need to access the state of a push button. Therefore, add a variable of type Button.

Now you connect the beans of PanelA (see Figure 72):

- ❑ Connect the items property of the OrgManager bean with the elements property of the OrgResultForm list box (1).
- ❑ Connect the actionPerformed event of the **List All Organization** button with the OrgManager select method (2). You can add a WHERE clause as the parameter of this connection. Because you want to select all the rows, there is nothing to enter. However, the connection remains dashed; you can enter a blank character as the SQL suffix to make the connection a solid line.
- ❑ To pass the selected row to PanelB connect the itemStateChanged event of the list box and the this method of the Org variable (3). Connect the selected object of the list box with the value parameter of the previous connection (4).
- ❑ Initially the **Update Selected Organization** button must be disabled. Open its properties and set the enabled property to false. The button must also be disabled after the list of organizations is retrieved. Connect the **List All Organization** button with the enabled property (or the setEnabled method) and set the parameter value to false (5).
- ❑ To enable the **Update Selected Organization** button when the user selects a row in the list box, connect the itemStateChanged event of the list box with the enabled method of the button. Set the parameter of this connection to true (6).
- ❑ To disable the **Update Selected Organization** button from PanelB, connect its this property with the this property of the button variable. The variable will make the button's attributes available in PanelB (7). The purpose of the button variable and this connection will become more clear when we complete PanelB.

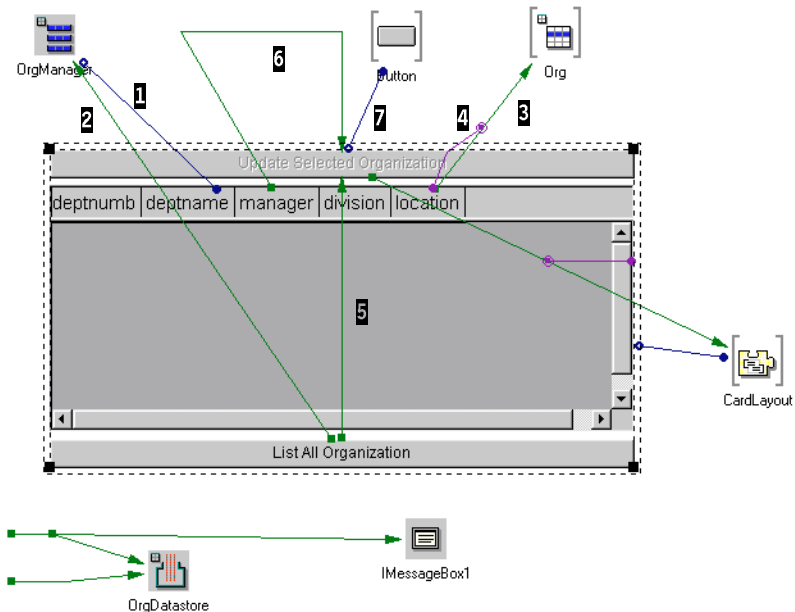


Figure 72. Organization Applet: Organization List Connections

Before you continue with the organization detail panel (PanelB), you can test PanelA by running the application. Check that this application indeed satisfies the PanelA requirements.

When you click the **Update Selected Organization** button, PanelB appears. Although the Org variable is loaded with the data from the selected row, its data is not displayed, because the connection between the Org Variable and the OrgForm has not been established.

Completing the Organization Detail Panel

The processing of PanelB requires the following connections (see Figure 73):

- ❑ To display the data of the Org variable, connect the Org variable (this) with the OrgForm (targetAsOrg) (1). In fact, after this connection is established, you can test the application and check that the data of the Org variable is displayed in PanelB.
- ❑ When the application switches back from PanelB to PanelA, the **Update Selected Organization** button in PanelA is still enabled. To disable this button, connect the **List All Organization** button (actionPerformed) in PanelB with the enable property of the button variable and set the parameter to false (2).

- ❑ To complete PanelB, you have to connect all actionPerformed events of the add, retrieve, update, and delete buttons to the related methods (add, delete, retrieve, and update) of the Org variable (3). No parameters are needed for these methods of the Org bean. You also have to connect the ExceptionOccurred event of each connection to the showException method of the messageBox and pass the event data (4).

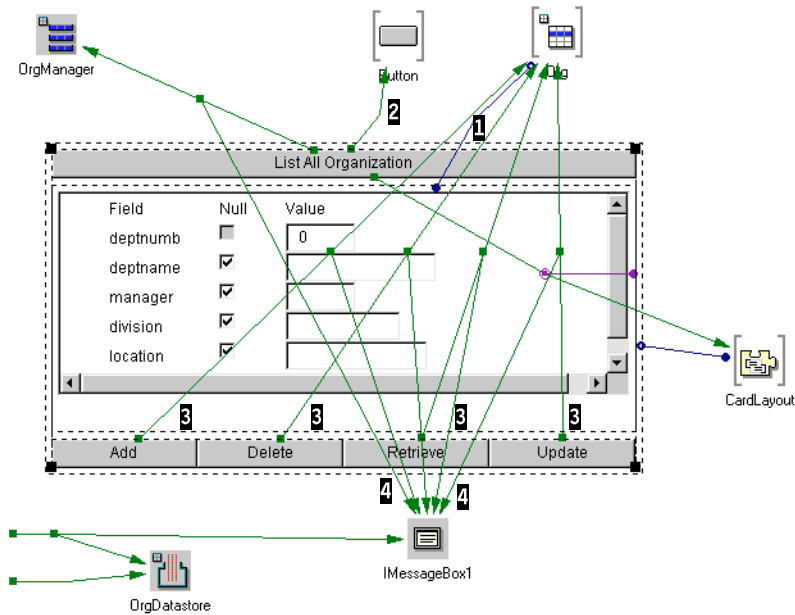


Figure 73. Organization Applet: Organization Detail Connections

This example concludes our introduction of Data Access Builder. We have created an applet accessing a relational database without writing a single line of code.

In Chapter 5, “ATM Application with Data Access Builder and JDBC,” on page 99, we use all the knowledge we have learned so far to write a more realistic application. We simulate an ATM to implement simple banking transactions.

Data Access Beans in Handwritten Programs

This example describes how you can incorporate beans generated by Data Access Builder into your handwritten programs. With this, you will appreciate not only the power of the tool but also its flexibility.

The example uses the datastore bean (OrgDatastore), the row manager bean (OrgManager), and the persistent object bean (Org) to perform SQL operations without writing SQL statements in a handwritten program.

To run this code, you have to perform the following steps:

1. Export the beans generated by Data Access Builder from VisualAge for Java into a directory in the CLASSPATH. Export creates a subdirectory with the name of the package used in VisualAge for Java (for example, SampleDax).
2. Another directory, SampleOrg, holds the SampleOrgDax program. Compile the program:

```
Javac SampleOrgDax.java
```

3. Run this command to execute the program:

```
java SampleOrgDax <userid> <password>
```

Figure 74 lists the SampleOrgDax program.

```
import SampleDax.*;
import java.lang.*;
import java.util.*;

public class SampleOrgDax
{
    public static void main(String args[])
    {
        try
        {
            // Create the Datastore and Manager objects
            OrgDatastore theOrgDatastore = new OrgDatastore();
            OrgManager   theOrgMgr       = new OrgManager();
            Org          theOrg          = new Org();

            // Connect to database
            System.out.println("Connecting to ORG database...");
            if ( args.length > 1 )
                theOrgDatastore.connect( args[0], args[1] );
            else
                theOrgDatastore.connect();
            System.out.println("...Connected");
            theOrgDatastore.setAutoCommit( false );
        }
    }
}
```

```
// List all orgs as they are
System.out.println("\nListing of all orgs, sorted by name:");
theOrgMgr.open( "ORDER BY DEPTNAME" );
while( theOrgMgr.fetchNext() )
{ System.out.println(theOrgMgr.element().toString()); }
theOrgMgr.close();

// Add a new department
theOrg.setDeptnumb((short)31);
theOrg.setDeptname("VAJE");
theOrg.setManager(new Short("26"));
theOrg.setDivision("Red Team");
theOrg.setLocation("San Jose");
theOrg.add();

// delete department 10
theOrg.setDeptnumb((short)10);
theOrg.delete();

// retrieve department 66 and change its location
theOrg.setDeptnumb((short)66);
theOrg.retrieve();
theOrg.setLocation( "Thornhill" );
theOrg.update();

// List all orgs again to show changes
System.out.println("\nListing of all orgs, sorted by name,
                    after changes:");
theOrgMgr.open( "ORDER BY DEPTNAME" );
while( theOrgMgr.fetchNext() )
{ System.out.println(theOrgMgr.element().toString()); }
theOrgMgr.close();

// Rollback the changes and disconnect
theOrgDatastore.rollback();
System.out.println("\nChanges rolled back");
theOrgDatastore.disconnect();

// All done
System.out.println("Done.\n");
}
// Catch all Throwables here and print out the information
catch (Throwable s)
{
    System.out.println( "Throwable caught:" );
    System.out.println( "message: " + s.getMessage() );
    System.out.println( "Stack Trace:" );
    s.printStackTrace();
}
}
```

Figure 74. Handwritten Program Using Data Access Beans

Here is the output of the program:

```
Connecting to ORG database...
...Connected

Listing of all orgs, sorted by name:
42.Great Lakes.100.Midwest.Chicago
10.Head Office.160.Corporate.New York
20.Mid AtlanticXX.10.Eastern.Washington
84.Mountain.290.Western.Denver
15.New England.50.Eastern.Boston
66.Pacific.270.Western.San Francisco
51.Plains.140.Midwest.Dallas
38.South Atlantic.30.Eastern.Atlanta
Listing of all orgs, sorted by name, after changes:
42.Great Lakes.100.Midwest.Chicago
20.Mid AtlanticXX.10.Eastern.Washington
84.Mountain.290.Western.Denver
15.New England.50.Eastern.Boston
66.Pacific.270.Western.Thornhill
51.Plains.140.Midwest.Dallas
38.South Atlantic.30.Eastern.Atlanta
31.VAJE.26.Red Team.San Jose

Changes rolled back
Done
```

The sample handwritten program uses many methods of the data access beans:

- ☐ Connect, disconnect, rollback, and setAutoCommit of the Org-Datastore bean
- ☐ Open, fetchNext, close, and element of the OrgManager bean
- ☐ Add, delete, retrieve, update, toString, and many of the set attribute methods of the Org bean

Use of the data access beans facilitates JDBC programming, even without the power of the Visual Composition Editor of VisualAge for Java.

Data Access Builder Advanced

Data Access Builder provides a number of additional functions that are not covered in detail in this book. In this section we provide an overview of some of those functions.

Sharing Mappings among Developers

One way to share Data Access Builder mappings among developers is through a .Data Access Builder file. To export a .Data Access Builder file, perform the following steps:

- ❑ In the Data Access Builder window, select *Export* from the *File* menu.
- ❑ Enter the directory and file name in the Export window and click on **OK** to export the mapping.

The Data Access Builder session with all its mappings will be exported to the Data Access Builder file. You can use *Import* in the *File* menu to read the .Data Access Builder file in another Data Access Builder session.

Another way of sharing mappings is through the interchange file (.dat) of the VisualAge for Java repository. A whole package (or even a project) can be exported from the Workbench into an interchange file and imported into another developer's repository.

Running Data Access Builder Stand-Alone

Data Access Builder can be run outside VisualAge for Java to create source beans that can be imported into the Workbench.

To start Data Access Builder by itself, issue this command:

```
ivjdata
```

The Startup window is displayed (Figure 75).

You can create a new mapping or resume work on an existing stored session. An existing stored session can be an exported mapping from a Data Access Builder session started from a Workbench package, or a session saved from a stand-alone run.

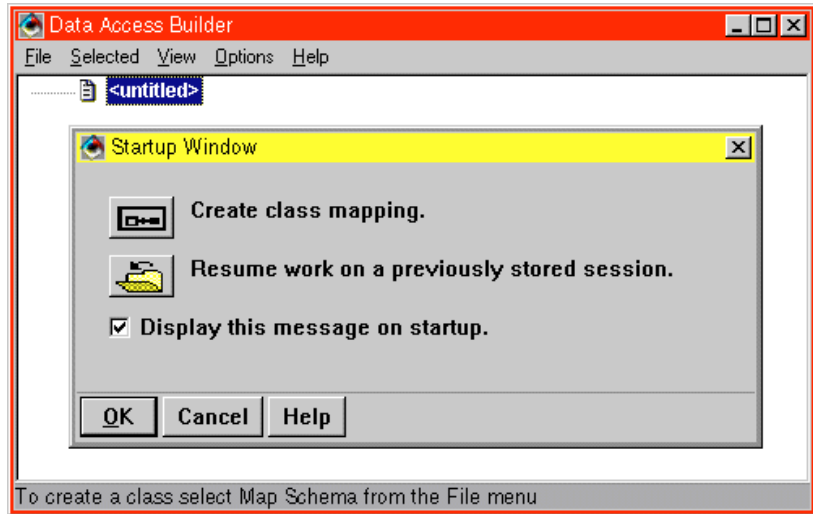


Figure 75. Data Access Builder: Startup Window

The mapping process itself is unchanged:

- Map a schema (by table, or enter an SQL statement)
- Tailor the bean (driver, database location)

When you are ready to generate the Java source code, use the *Options* pull-down to specify the directory where the code is generated. Select *Generate All* in the *File* pull-down to generate the beans. The code is written into a subdirectory with the name of the mapping.

Save the session in a user-defined directory, using *Save* in the *File* pull-down. The assigned name is displayed in the Data Access Builder window (Figure 76).

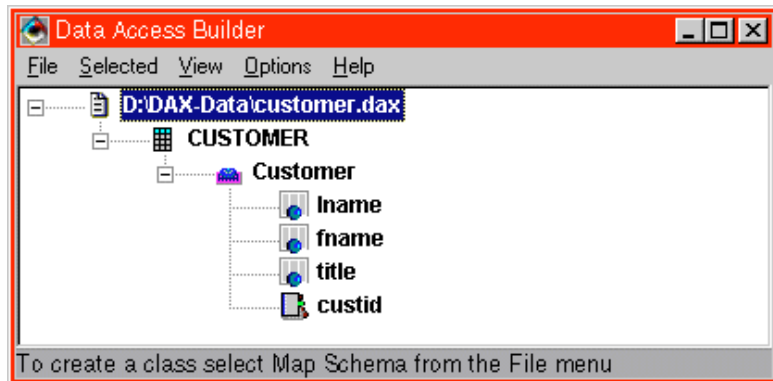


Figure 76. Data Access Builder: Save Session

The next time you use Data Access Builder you can resume work on a saved or exported session by selecting *Resume work on a previously stored session* in the Startup window, or by selecting *Open* in the *File* pull-down.

Import the generated beans into a project in VisualAge for Java. A package with the name of the mapping is generated automatically. The beans can be used in the same way as if they were generated directly into the IDE from Data Access Builder.

Note: The session is not stored in the repository. Data Access Builder must be started manually to update the mapping and regenerate the beans.

Interesting Methods of the Manager Bean

The manager bean (<class>Manager) contains two interesting methods: fill and append.

Both methods let you specify the number of rows to be retrieved. The difference between fill and append is that the latter does not overwrite the previously filled vector but rather appends the next rows at the end of the vector.

All the methods that create a result set cause the items property of the manager bean to fire. This is especially useful in the visual programming environment.

Eliminating Attributes from the Mapping

Occasionally a mapping will provide you with more attributes than you actually require. You can eliminate these to simplify the interface of your bean by using the *Attributes...* option from the pop-up menu of the bean in the mapping.

Select an attribute that you want to modify. Click on **Delete Attribute** to remove it from the mapping. Click on **Properties** to modify the attribute name or data type.

Customized SQL Statements

The persistent object bean can be extended with customized (user-defined) SQL statements before the beans are generated.

From the pop-up menu of the bean in the mapping, select *Methods...* Click on the **Add** button under *Methods* with customized SQL statements.

Enter an SQL statement, using question marks as place holders for any values to be substituted at run time (so-called host variables), and a method name. You can use INSERT, DELETE, or UPDATE statements, as well as a single-row SELECT statement. (Selects of multiple rows are added as methods to the <class>Manager bean).

Validate the SQL statement; for each host variable (denoted by ?) and retrieved column (if you entered a single-row SELECT statement) a parameter is generated.

You can change the mapping of host variables by giving a user-friendly name to the parameter or by mapping it to an attribute of the bean.

The mapping of returned columns of a single-row SELECT statement can be changed from parameter to an attribute of the bean, or to the return value of the method. Only one column can be the return value.

The remapping process consists of these steps:

- ☐ Select one of the parameters.
- ☐ Select the appropriate radio button for the type of mapping.
- ☐ For a parameter, update the item's name and optionally its type.
- ☐ For an attribute, tie the item to a particular attribute of the bean.
- ☐ For a return value, select the data type.
- ☐ Click on the **Modify** button.

For an example of a customized SQL statement, refer to “Adding User-Defined Methods” on page 106.

Encapsulating an SQL Search Predicate

The manager bean can be extended with customized (user-defined) SQL SELECT statements before the beans are generated.

From the pop-up menu of the bean in the mapping, select *Manager Methods...* Click on the **Add** button under *Methods* with SQL predicates.

Enter an SQL predicate, for example, a WHERE clause, using question marks (host variables) as place holders, and validate the statement.

To modify the parameters that are generated for the host variables, assign user-friendly names and appropriate data types as described in “Customized SQL Statements” on page 94.

Asynchronous Processing

Background thread support is provided so that you can execute long-running methods asynchronously. You call `setAsynchronous(true)` before invoking methods that execute SQL statements.

When you use asynchronous processing, a method is not necessarily complete when control returns to the program; rather it is complete when the background thread reports a method complete event. Only one asynchronous thread per object is allowed at one time.

Working with Stored Procedures

Stored procedures should be defined in the database system before you use them in Data Access Builder. However, you can also add their definitions manually.

Like the customized SQL statements, stored procedures can be invoked as persistent object methods or manager methods. Parameters of stored procedures can be defined as input or output.

Let us look at the process of defining a method for a stored procedure:

- ☐ Open the Methods window by selecting *Methods...* in the pop-up menu.
- ☐ Click on **Add** under Methods with stored procedure calls.
- ☐ Click on **Show Procedures** in the Add Stored Procedure Call window.
- ☐ In the Stored Procedures List window, click on **Get Stored Procedures**, to retrieve the definitions from the database, or on **Add** to define one stored procedure manually.
- ☐ To define a stored procedure, define all the parameters as input or output with their SQL names.
- ☐ Select a stored procedure in the Add Stored Procedure Call window to display its parameters.

- ❑ Change the mappings of the parameters. Input parameters can be mapped as parameters or attributes of the bean. Output parameters can be mapped as parameters, attributes, or as a return value. Click on **Modify** for each change.
- ❑ Close the window when finished. Close the Methods window as well.
- ❑ Generate the data access beans.

The methods for stored procedures are invoked in the application in the same way as other SQL encapsulating methods. Connect an event to the method and pass the input parameters.

Handling of the results is a little more tricky. Each output parameter is actually mapped to an array, not to a single value. In many cases you need a script (a user-defined method in the application) to retrieve the results from the array.

Application Design Considerations

Once your database-related beans have been generated, they are ready to be used from within your business logic. They can also be used directly from within your user interface beans, but we generally recommend against that in production-level code.

Production programs should be partitioned into user interface, business logic, and persistence segments, and the user interface portion typically should not have direct knowledge of persistence issues. This approach makes the program more robust because the persistence layer can be replaced without any impact on the user interface layer (and vice versa).

Another approach to consider is to have your database-related beans (and any business logic beans that use them) reside on a server machine and your user interface beans reside on a client machine. This approach can improve performance because your business logic and database access run on a dedicated server machine (where the database may also reside), and the amount of code needed on the client machine is reduced.

Java's RMI technology can provide the communication layer between the client and server beans. RMI beans, like data access beans, can be generated using the Enterprise Edition of VisualAge for Java. For more information about generating and using RMI beans in VisualAge for Java, refer to Chapter 6, "Remote Method Invocation and RMI Access Builder".

5

ATM Application with Data Access Builder and JDBC

In Chapter 4, “Data Access Builder”, you discovered the power of the tool, and you constructed a first sample application that used the beans generated by Data Access Builder to manage data stored in the DB2 sample database. Now it is the time to use this new knowledge to build something more complex—the ATM application.

In the first part of this chapter, we review the ATM application requirements and, on the basis of these requirements, design the application.

In the second part, we implement the design. We use Data Access Builder to create the data access beans, and we use the Visual Composition Editor to create the ATM application.

Designing the ATM Application

The design of the ATM application is based on Chapter 2, “Sample ATM Application and Database”.

We assume that the ATM database exists, as described in “ATM Database” on page 29. We implement the GUI described in “ATM Application Requirements” on page 18 and shown in Figure 13 on page 19.

We use a simple object model that is based on the underlying database (Figure 77).

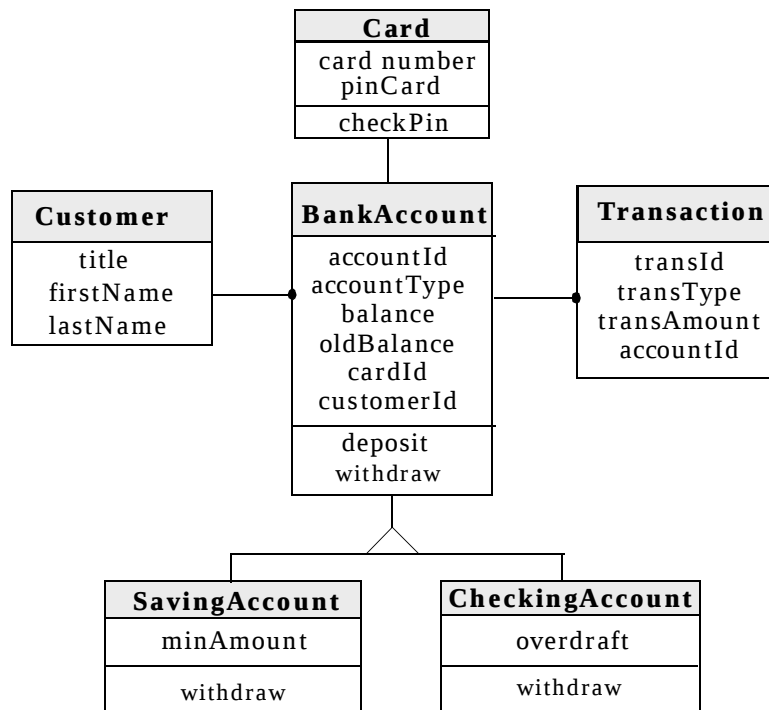


Figure 77. Object Model of the ATM Application

We did not actually design this model. The purpose of this chapter is to explain how we can use the Visual Composition Editor to create an application based on beans generated by Data Access Builder from an existing relational database.

In Chapter 7, “ATM Application with RMI” we use a more realistic design with a business model and a controller for the same application.

Building the ATM Application

After the application design phase, we are ready to start with the implementation phase.

In our implementation we can identify four steps that we have to follow to build the ATM application:

1. Construct database classes
2. Construct business logic classes
3. Construct user interface classes
4. Implement application flow

Because the ATM application uses a relational database to store and retrieve the necessary information, we use Data Access Builder to create the beans that access our database tables.

When we have built our database interface, we need to identify the classes required by our business logic design and the classes that will act as user interface.

In the end, we construct the application logical flow by connecting objects of the previously constructed classes, so that they can communicate with each other by sending and receiving messages.

Database Classes

In this section we describe all the database classes created with Data Access Builder and explain how we generate them.

Create a project named JDBC in the Workbench, and add two packages, ATMDax and ATMApplication. You generate all database classes in the ATMDax package. The ATMApplication package contains the application classes.

PIN Validation

The first information that the ATM application retrieves from the database is the customer data and the PIN, starting from the card ID that is used at the ATM. The customer information is needed for the greeting, and the PIN is needed for validation (see “ATM Application Requirements” on page 18).

Based on the ATM database structure (see “Database Design” on page 21), to retrieve all of the required data we have to join two ATM tables: Customer and Card. Therefore, we create a mapping schema starting from a join and use the following SQL statement:

```
SELECT A.PIN, B.TITLE, B.FNAME, B.LNAME
FROM ATM.CARD A, ATM.CUSTOMER B
WHERE A.CARDID = ? AND A.CUSTID = B.CUSTID
```

In this statement the ? represents the parameter, that is, the card ID entered by the user. We name the mapping schema **Pin-CustInfo**.

Now let us show you how we can construct this mapping schema from the above SQL statement, using Data Access Builder.

When we start Data Access Builder to create the mapping schema, the dialog shown in Figure 78 is displayed.

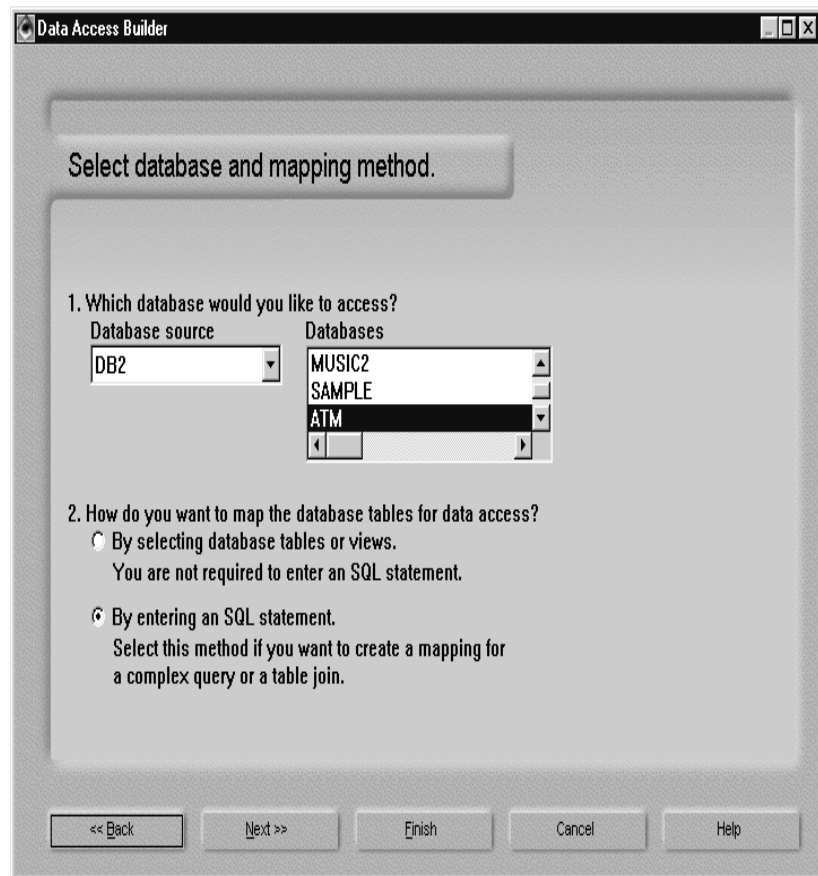


Figure 78. Data Access Builder SmartGuide Dialog

In this dialog we select the ATM database and the mapping method. We check the radio button indicating that we want to map the schema by entering an SQL statement. We click on the **Next** button to continue with the dialog shown in Figure 79.

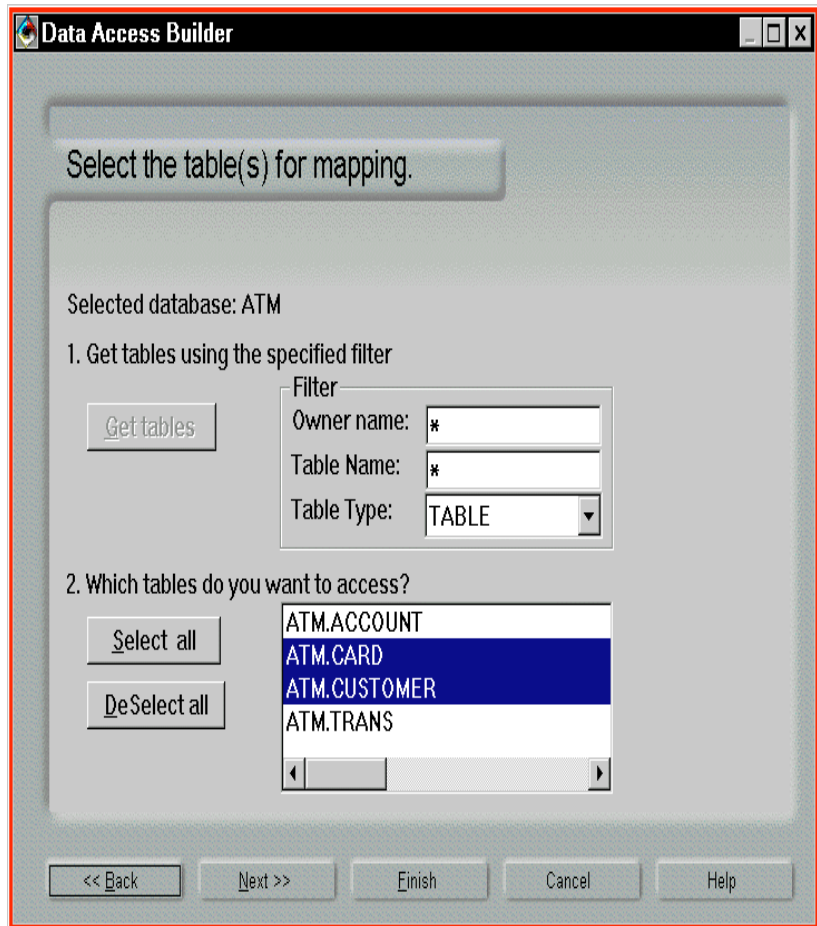


Figure 79. Selecting the Tables for Mapping

We click on the **Get tables** button to display all tables in the list box. We select the ATM.CARD and ATM.CUSTOMER tables in the list box as the basis for the SQL statement.

We click the **Next** button to continue with the dialog shown in Figure 80.

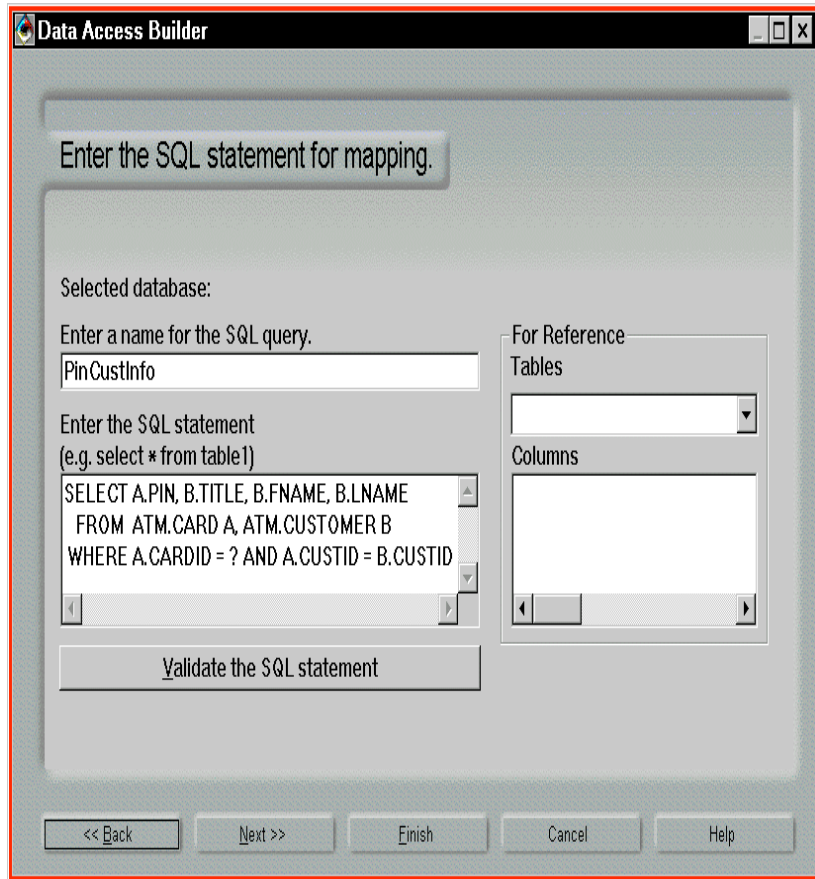


Figure 80. Validating the SQL Statement

In this dialog we enter the name of the SQL query and the complete SQL statement. We validate the SQL statement by clicking on the **Validate the SQL statement** button.

In the next dialog we can enter a description for the mapping schema. This optional description is carried forward as Java comments into the generated data access beans.

Click on the **Finish** button to terminate the SmartGuide dialog.

The Database Access Builder window now contains the mapping schema icon that Data Access Builder has created from the SQL statement.

We will follow this procedure for all mapping schemas of the ATM application.

List of Accounts

When the PIN is successfully validated, the system retrieves all the account IDs that belong to the customer.

To retrieve all the account IDs, we need to create another schema, which is based on the account table. This schema retrieves from the account table all the accounts related to the customer. We create this mapping class, using the following SQL statement:

```
SELECT ACCID FROM ATM.ACCOUNT WHERE CARDID = ?
```

In this statement the parameter is the card ID. We call this schema **QueryAccId**.

Account Information

When the ATM application displays all the account IDs, the customer selects one account to start the debit and credit transactions. When an account is selected, the ATM application shows the data related to the specified account, that is, account ID, account type, and account balance.

To retrieve the account data, we have to map the ATM account table. We do not enter an SQL statement; instead we just map the account table itself. The bean generated by Data Access Builder retrieves the required record, using the selected account ID. (See “Building a JDBC Application” on page 61 for an explanation of how to create a mapping from a table). The default name of this mapping is **ACCOUNT**.

Transaction History

The ATM application provides information about all transactions related to the specified account. This transaction history includes, for each transaction, a transaction ID, an account ID, a transaction type, and a transaction amount.

All of this information is stored in the Transaction table. To handle the transaction list associated with an account, we have to create another mapping schema. This schema is mapped based on the following SQL statement:

```
SELECT * FROM ATM.TRANS WHERE ACCID = ?
```

The parameter in this SQL statement is the account ID. We call this schema **TransAcc**.

After creating all the database beans that we have discussed so far, our Data Access Builder window contains the mappings (data-base beans) shown in Figure 81.

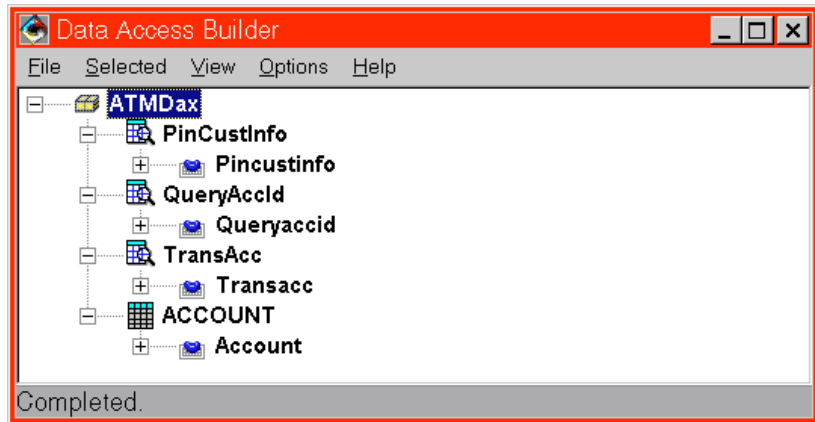


Figure 81. Database Access Builder Window: Mappings

At this point we can open the mapped beans (select *Attributes...* from the pop-up menu) to delete unused columns or change the names of the properties to make them more readable than the column names in the database. We decided to leave the beans unchanged.

Adding User-Defined Methods

The customer can also perform new transactions, such as, deposit or withdrawal. When a transaction is completed, the system updates the transaction list by adding it to the transaction table. Therefore, the Transacc bean must be able to add a new transaction record to the Transaction table when a deposit or withdrawal operation occurs.

Data Access Builder allows us to add user-defined methods to the mapping before generating the beans. Therefore, we add an **addTransaction** method to the TransAcc bean. This method inserts a new row into the Transaction table, based on the database structure.

We create the unique transaction ID property, using the Timestamp database function. The account ID property is related to the account of the transaction. The transaction type property comes from the operation type performed by the customer (D for deposit, W for withdrawal). The transaction amount property is the amount specified by the customer.

The `addTransaction` method performs the following SQL statement:

```
INSERT INTO ATM.TRANS VALUES(CURRENT_TIMESTAMP,?,?,?)
```

The three parameters of the statement are account ID, transaction type, and transaction amount; therefore, the method that Data Access Builder generates requires three input parameters.

Let us now explain in detail how we add this new method to the `Transacc` bean.

In the Data Access Builder window we highlight the bean where we want to add a method. From the pop-up menu (right mouse button) select *Methods...* (Figure 82).

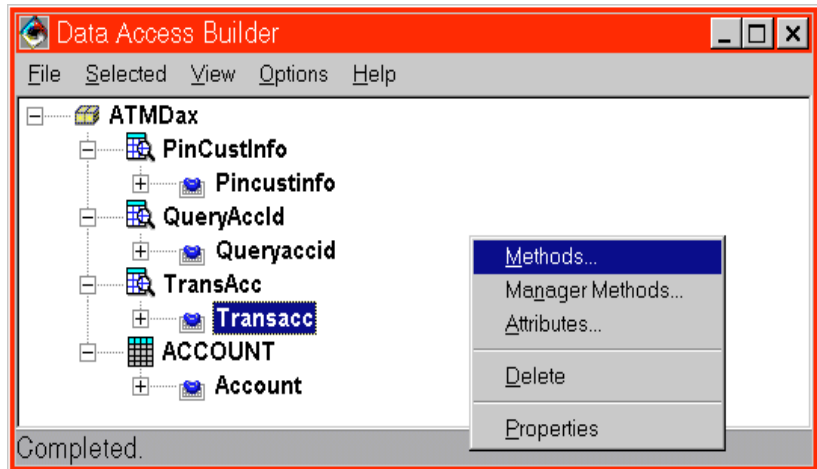


Figure 82. Context Menu of a Dax Bean

In the *Methods* window, we click on the **Add** button to add a method that uses a customized SQL statement (Figure 83).

We enter the SQL statement that the method has to perform and click on the **Validate** button to validate the statement. We can change the names of the parameters by selecting each parameter in the list, overtyping the name in the Name field, and clicking on the **Modify** button.

For the `addTransaction` method we replace the default `param1` with `accountId`, `param2` with `transactionType`, and `param3` with `transactionAmount`. Later, when we connect to the generated bean in the Visual Composition Editor, we can recognize the parameters by their meaningful names.

Transacc - Add Customized SQL Statement

SQL Statement: Validate

INSERT INTO ATM.TRANS
VALUES (CURRENT TIMESTAMP, ?, ?, ?)

Method Name:

☒ Throw exception if multiple rows affected

Parameters

Name	Type	Mapped To	SQL Name	SQL Type
✓ accountId	String	Parameter	param1	CHAR[8]
✓ transactionType	String	Parameter	param2	CHAR[1]
✓ transactionAmount	BigDecimal	Parameter	param3	DECIMAL[8,2]

Parameter Type: SQL Name: SQL Type:

Mapped To: ☒ Parameter ☐ Attribute ☐ Return Value ☐ None

Name: Type:

Modify OK Cancel Help

Figure 83. Validating the SQL Statement and Setting Parameter Names

To add the new method to the bean, we have to assign it a name, in our case, `addTransaction`. We click on the **OK** button to add the method to the Transacc bean.

Generating the Data Access Beans

We open the four mapping beans, set the table qualifier to `ATM`, the driver to `COM.ibm.db2.jdbc.net.DB2Driver`, and the URL to `jdbc:db2://hostname:8888/ATM`. We set the host name to *localhost* to run on a stand-alone machine.

We generate the beans into the Workbench, using *Save and Generate All* in the *File* pull-down.

Business Logic Classes

In this section we introduce the business logic classes used by the ATM application. These classes do not have a visual interface, but they enable us to build the application logic.

Our implementation requires the following objects:

- ☐ Card
- ☐ BankAccount
- ☐ SavingAccount
- ☐ CheckingAccount

The Customer and Transaction objects (see Figure 77 on page 100) are never materialized. The information retrieved from the database into the data access beans is used directly in the GUI.

Before we describe in detail how we implement each class, it is necessary to understand their responsibilities, and how they are related to each other.

In our implementation, Card is the class that knows the card number entered by the user and the PIN related to that card number. Starting from this information, it must perform the PIN validation when the user enters the PIN. After validation, the Card sends a successful or unsuccessful message.

BankAccount, SavingAccount, and CheckingAccount are closely related to each other. In fact, the BankAccount class represents the generic bank account, with all of the features of a bank account, such as account ID, balance, and customer ID. SavingAccount and CheckingAccount inherit from BankAccount, but each of them has additional information and behavior. The SavingAccount class contains the information related to the minimum amount that it can reach; the CheckingAccount, instead, contains its overdraft value. In other words, the instances of the SavingAccount and CheckingAccount classes represent the real accounts of the customer.

When a customer uses the ATM application to perform a withdrawal transaction, different answers can come from the system, depending on the kind of account. The SavingAccount class performs the withdrawal transaction only if the balance, after the transaction, is still greater than the minimum amount; the CheckingAccount class checks that a resulting negative balance is higher than the overdraft amount.

When a customer requires a deposit transaction, both the SavingAccount and CheckingAccount classes have the same behavior, that is, they increase the balance by the deposit amount.

Let us describe how we implement the business classes, so that you can do the same in your project.

Remember, you have to create all nonvisual classes in the ATMApplication package.

The business logic classes are implemented as Java beans, that is, the features (properties, methods, and events) are defined on the BeanInfo page of the class. We create the properties as read/write with get and set methods and define that they are *bound*, that is, they fire the PropertyChange event whenever the value changes.

Card Class

We implement the Card class as a nonvisual bean with the following properties:

- ❑ `cardNumber`, of type `String`, bound
- ❑ `pinCard`, of type `String`, bound

The internal attribute name for a property has the prefix *field*, followed by the property name with the first letter in upper-case. For example, the `pinCard` property is represented by the `fieldPinCard` attribute, and the get and set methods are called `getPinCard` and `setPinCard`.

The Card class has to send messages to other objects about the result of the PIN validation. We implement this behavior as two events, `pinCheckedOk` and `pinCheckedNotOk`.

To define and fire an event, we perform the following steps:

- ❑ We create a new event listener interface, using the New Event Listener SmartGuide (select *New Listener Interface* in the *Features* pull-down). See Figure 84.
- ❑ In the first page of the dialog, we enter the name of the event, `pinCheckedOk` (or `pinCheckedNotOk`), and click on **Next**.
- ❑ In the second page, we enter the name of the method that the listener class has to implement, `handlePinCheckedOk` (or `handlePinCheckedNotOk`), and click on **Add**, then on **Finish**.

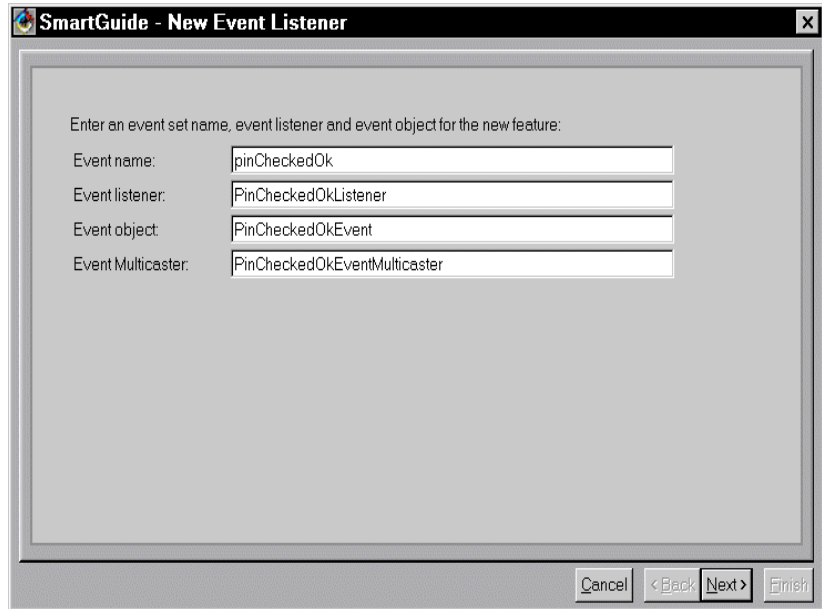


Figure 84. Defining an Event with an Event Listener

The system automatically generates the event and interface classes and the supporting methods for us:

- PinCheckedOkEvent class
- PinCheckedOkListener interface
- fireHandlePinCheckedOk method
- addPinCheckedOkListener method
- removePinCheckedOkListener

All we have to implement is the logic for when to fire the events. On the BeanInfo page we add a new method feature to the Card class and call it checkPin. The checkPin method compares the pinEntered parameter with the pinCard property and notifies the caller by firing the pinCheckedOkEvent or the pinCheckedNotOkEvent:

```
public void checkPin (String pinEntered)
{
    if (getPinCard().trim().equals(pinEntered.trim()))
        fireHandlePinCheckedOkEvent(new PinCheckedOkEvent(this));
    else
        fireHandlePinCheckedNotOk(new PinCheckedNotOkEvent(this));
    return;
}
```

BankAccount Class

We implement the `BankAccount` class as an abstract class.

`SavingAccount` and `CheckingAccount` are subclasses of `BankAccount`. In fact, we want to reuse the attributes and methods of the `BankAccount` class in the `SavingAccount` and `CheckingAccount` classes. In the ATM application we create only instances of `SavingAccount` or `CheckingAccount`, that is, the real accounts that belong to a customer.

The properties of the `BankAccount` class reflect the data stored in the account table of the ATM database, and two additional properties to hold the old balance and the decoded account type:

- ❑ `accountId`, of type `String`
- ❑ `accountType`, of type `String`
- ❑ `balance`, of type `double`
- ❑ `cardId`, of type `String`
- ❑ `customerId`, of type `String`
- ❑ `oldBalance`, of type `double`
- ❑ `typeDecod`, of type `String` (see below)

To initialize the attributes of a bank account object we add a constructor to the `BankAccount` class:

```
public BankAccount (String accType, String accId, String cardId,
                    String custId, java.math.BigDecimal bal) {
    setAccountId (accId);
    setAccountType(accType);
    setCardId(cardId);
    setCustomerId(custId);
    setBalance (Double.valueOf(bal.toString()).doubleValue());
    setOldBalance(getBalance());
}
```

Notice that we pass the `bal` parameter as `BigDecimal` because the `BankAccount` object is initialized from the database and the balance column of the account table is defined as decimal.

We want to display the account type as a word, `Checking` or `Savings`, not just as code `C` or `S`. For this purpose we define a property, called `typeDecod`, as a `String`, and a method, called `DecodAccountType`, that converts the database code into the property value:

```
public void DecodAccountType (String type) {
    if (type.trim().equals("C"))
        setTypeDecod("Checking");
    else
        setTypeDecod ("Savings");
}
```

Because the deposit transaction is not dependent on the account type, we can implement the deposit method feature in the `BankAccount` class and use the same implementation for `SavingAccount` and `CheckingAccount`. This method stores the current balance in the `oldBalance` property and then adds the amount entered by the user:

```
public void deposit (String amount) {
    double amnt = Double.valueOf( amount.toString()).doubleValue();
    setOldBalance(balance);
    setBalance(balance + amnt);
}
```

The withdrawal transaction is dependent on the account type. Therefore, we define the `withdraw` method feature in the `BankAccount` class as abstract and provide the implementation in the `SavingAccount` and `CheckingAccount` subclasses:

```
public abstract void withdraw (String amount);
```

`BankAccount` fires two events to inform the caller whether the withdrawal transaction is allowed or not. If a withdrawal is not allowed, the ATM application can display a message to the user. We implement this behavior in the same way we implemented it for the `Card` class:

- ❑ We create a new event listener interface for the withdrawal-Failed event, with the `handleWithdrawalFailed` method.
- ❑ We create a new event listener interface for the withdrawal-Done event, with the `handleWithdrawalDone` method.

After code generation we find the `fireHandleWithdrawFailed` and `fireHandleWithdrawDone` methods in the `BankAccount` class. The `SavingAccount` and `CheckingAccount` classes will use these methods in their own `withdraw` method to fire one of the two events.

At last, we need to add a converter method feature to the `BankAccount` class to convert a double value to a `BigDecimal` value. The balance property is stored as double in the `BankAccount` class, but the database implementation needs a `BigDecimal` value:

```
public java.math.BigDecimal converter (double arg) {
    Double f = new Double(arg);
    return java.math.BigDecimal.valueOf(f.longValue());
}
```

CheckingAccount Class

CheckingAccount is a subclass of the BankAccount class. In addition to the function of the basic account class, it provides overdraft protection.

We add a new property, called overdraft, as a double value.

In the constructor of the CheckingAccount class, we initialize the overdraft property with the value from the ATM account table:

```
public CheckingAccount (String accType, String accId,
                        String cardID, String custId,
                        java.math.BigDecimal bal,
                        java.math.BigDecimal over) {
    super(accType, accId, cardID, custId, bal);
    setOverdraft (Double.valueOf(over.toString()).doubleValue());
}
```

The CheckingAccount class has to implement its own withdraw method. This method checks whether the account balance can be updated on the basis of the balance and overdraft properties. The method notifies other objects of the result of the withdrawal transaction, using one of the events defined in the BankAccount class:

```
public void withdraw (String amount) {
    double amnt = Double.valueOf(amount).doubleValue();
    if ( getBalance() + getOverdraft() > amnt )
    { setOldBalance (getBalance());
      setBalance (getBalance() - amnt);
      fireHandleWithdrawalDone(new WithdrawalDoneEvent(this)); }
    else
      fireHandleWithdrawalFailed(new WithdrawalFailedEvent(this));
}
```

SavingAccount Class

SavingAccount is a subclass of the BankAccount class. In addition to the function of the basic account class, it has to maintain a minimum balance. We add a new property, called minAmount, as a double value.

In the constructor of the SavingAccount class, we initialize the minAmount property with the value from the ATM account table:

```
public SavingAccount (String accType, String accId,
                      String cardID, String custId,
                      java.math.BigDecimal bal,
                      java.math.BigDecimal min ) {
    super(accType, accId, cardID, custId, bal);
    setMinAmount (Double.valueOf(min.toString()).doubleValue());
}
```

The `SavingAccount` implements the `withdraw` method, considering the `minAmount` property value, and fires one of the events defined in the `BankAccount` class:

```
public void withdraw (String amount) {
    double amnt = Double.valueOf(amount).doubleValue();
    if (getBalance() - amnt > getMinAmount() )
    { setOldBalance (getBalance());
      setBalance (getBalance() - amnt);
      fireHandleWithdrawalDone(new WithdrawalDoneEvent(this)); }
    else
      fireHandleWithdrawalFailed(new WithdrawalFailedEvent(this));
}
```

User Interface Classes

Now it is the time to construct the user interface for the ATM application. Based on the requirements (see “ATM Application Requirements” on page 18), we have to build four visual classes to implement the application flow:

- ❑ `CardPanel`
- ❑ `PinPanel`
- ❑ `SelectAccountPanel`
- ❑ `TransactionPanel`

All four classes inherit from the `java.awt.Panel` class, and we will often refer to them simply as *panels*. Let us now examine each of these classes.

CardPanel Class

When the ATM application starts, an instance of the `CardPanel` class is created. The function of this panel is to allow the customer to enter the card number. In fact, this panel simulates the real ATM behavior, where the user slides the ATM card through a card reader.

From the graphical point view, this panel is very simple. It contains a label, a text field and two push buttons. To construct this panel, we use the `GridBagLayout` manager to control the location of the components when the panel is resized (Figure 85).

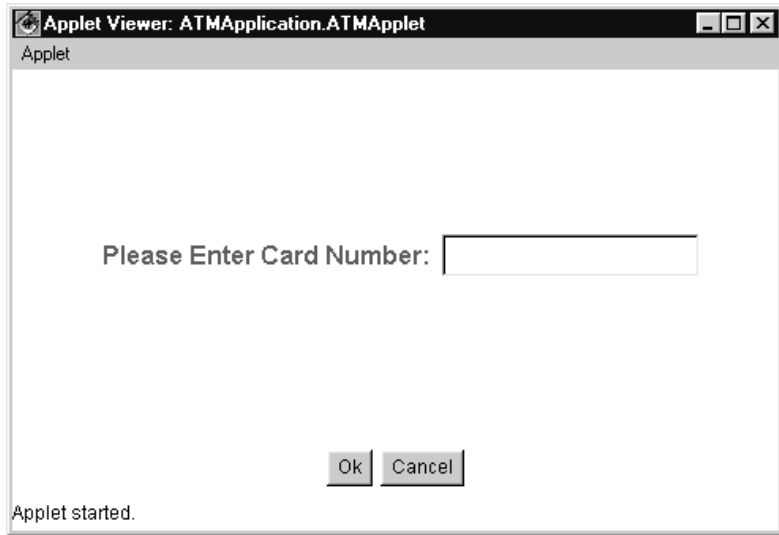


Figure 85. Layout of the CardPanel

We assume that you know how to construct such a GUI. Therefore, we explain in detail only how we build it from a functional point of view, that is, which nonvisual model objects we use and how all the objects are connected.

There are four major steps to complete the implementation of the panel:

1. Add the nonvisual beans that represent the business logic and the database.
2. Define an auxiliary property and method to format the customer name.
3. Connect the visual and nonvisual beans.
4. Promote some of the features to make them accessible in the ATM application (where the panel will be embedded).

Figure 86 shows the design of the panel inside the Visual Composition Editor.

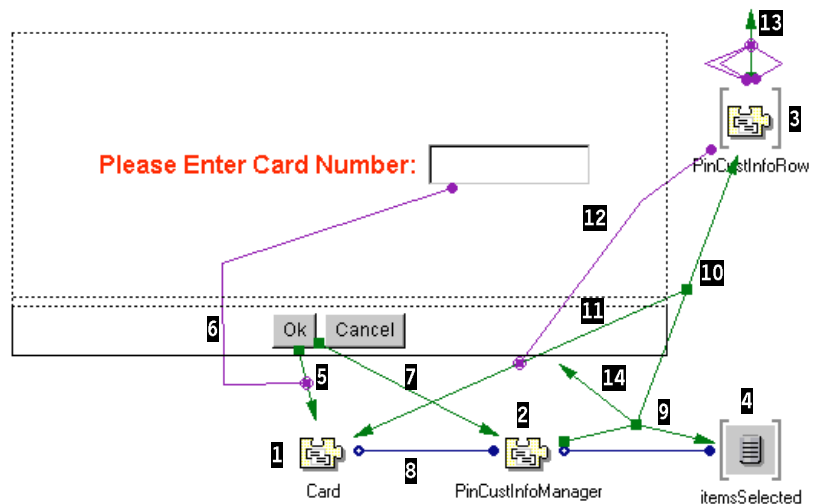


Figure 86. Visual Composition Editor: CardPanel

Nonvisual Beans

We start by adding three nonvisual beans to the free-form surface:

- ❑ A Card bean from the ATMApplication package, called Card **(1)**
- ❑ A PincustinfoManager bean from the ATMDax package, called PinCustInfoManager **(2)**
- ❑ A Pincustinfo bean variable from the ATMDax package, called PinCustInfoRow **(3)**

Auxiliary Property and Method

We define a String property, called `customerInfo`, and a method, called `createCustomerName`, in the `CardPanel`. The `createCustomerName` method formats the customer's full name from the title, first name, and last name:

```
public void createCustomerName (String title, String fname,
                               String lname) {
    String customer = title.trim()+" "+fname.trim()+" "+lname.trim();
    setCustomerInfo(customer);
}
```

The input strings for this method will be retrieved from the customer table in the ATM database.

Connections

Applications are driven by events. The connection diagram is a visual representation of the events that the application handles.

- ❑ We tear off the items property from the PinCustInfoManager bean and call it itemsSelected (4). The selectedItems property is a Vector. By tearing it off, we can access the methods of this Vector directly. We do that when we select the first record retrieved by the PinCustInfoManager.
- ❑ The Card bean stores the value entered by the user in its cardNumber property. We connect the actionPerformed event of the **Ok** button with the cardNumber property of Card (5) and pass the text attribute of the text field as a parameter (6).
- ❑ We use the PinCustInfoManager to retrieve the PIN value and the customer data from the database when the card number is entered. We connect the actionPerformed event of the **Ok** button with the select method of PinCustInfoManager (7). We connect the cardNumber property of Card with the param1 property of PinCustInfoManager; this is the host variable that is inserted into the SQL select statement (8).
- ❑ We store the one record retrieved by PinCustInfoManager in PinCustInfoRow. (We are sure that the PinCustInfoManager select function returns one record because of the unique keys.) We connect the selectComplete event of PinCustInfoManager with the elementAt method of itemsSelected (the Vector) and set the parameter to 0 (9). In this way, the elementAt method returns the first element selected by PinCustInfoManager. We connect the normalResult of the previous connection (elementAt method) with the this property of PinCustInfoRow (10).
- ❑ We store the PIN value retrieved from the database in the Card bean. We connect the normalResult of the previous connection with the pinCard property of Card (11) and pass the pin property of the PinCustInfoRow as a parameter (12).
- ❑ Now we can construct the customer's full name from the PinCustInfoRow. We connect the this event of PinCustInfoRow with the createCustomer method of the CardPanel bean; this is an event-to-script connection (13). We set the three parameters of the method by connecting the properties title, lname, and fname of PinCustInfoRow.
- ❑ In case the card number is invalid, we set the customer's name to an error message. We connect the exceptionOccurred of connection 9 with the createCustomer method of the CardPanel bean and set the three parameters to "Error:", "Invalid card-", and "Press Cancel" (14).

Promote Features

We need to make some features of the card panel available to the ATM application. Therefore:

- ❑ We promote the actionPerformed events of the **Ok** and **Cancel** buttons and name the promoted features OkClicked and CancelClicked.
- ❑ We promote the this property of Card and call it card, and the text property of the text field and call it cardIdFieldText.

PinPanel Class

The PinPanel class allows the user to enter the PIN number associated with the ATM card. The PinPanel is built by using six labels, one text field for the PIN number, and two push buttons. We define the layout manager as GridBagLayout. Figure 87 shows the PinPanel GUI.

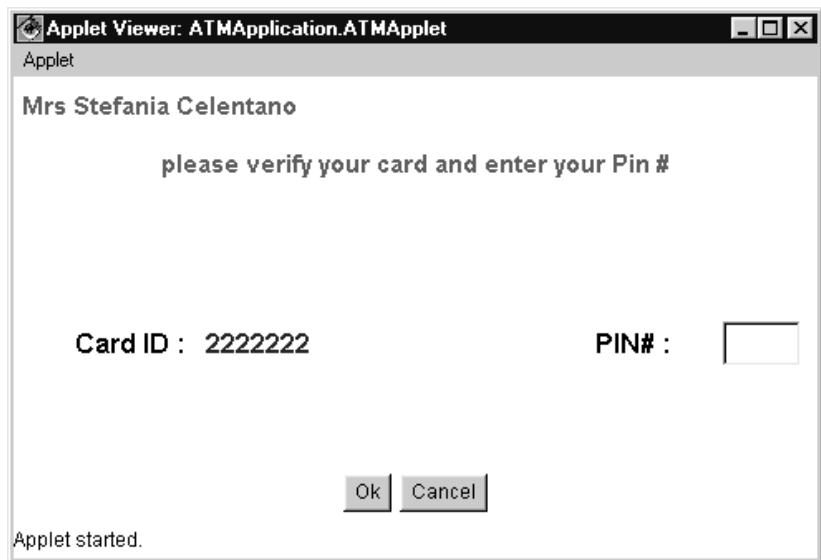


Figure 87. Layout of the PinPanel

Three of the labels display fixed text (“please verify your card and enter your PIN #,” “Card ID,” and “PIN#”) and three labels display variable text. We name the variable labels customerNameLabel, cardNumLabel, and errorMessage. The error message is located above the push buttons, but it is initialized with blanks.

To complete the implementation of the PinPanel, we have to connect some beans and promote some features.

Figure 88 shows the design of the panel inside the Visual Composition Editor.

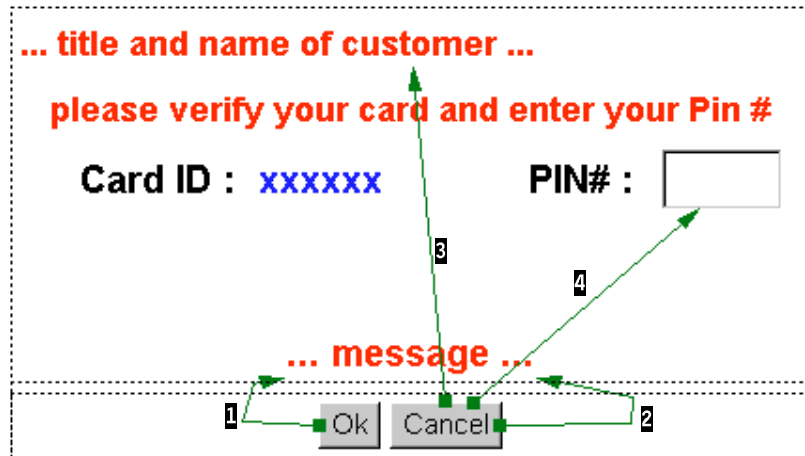


Figure 88. Visual Composition Editor: PinPanel

Connections

The **Ok** and **Cancel** buttons have to clear the labels with varying text (customer name and error message) and the PIN text field:

- ❑ We connect the actionPerformed event of the **Ok** button with the text property of the error message label (errorMessage) and set the parameter to a blank string (1).
- ❑ We do the same for the **Cancel** button (2).
- ❑ We connect the actionPerformed event of the **Cancel** button with the text property of the customer full name label (customerNameLabel) and set the parameter to a blank string (3).
- ❑ We connect the actionPerformed event of the **Cancel** button with the text property of the PIN number text field and set the parameter to a blank string (4).

Promote Features

We have to promote all the features of the PinPanel that have to be accessed when the PinPanel is embedded in the application. This customer full name label and the card ID label have to be set when the panel is displayed, and the error message label, after PIN validation. We also have to access the PIN number text field and the two push buttons:

- ❑ We promote the text property of the customer full name label as `customerNameLabelText`, the text property of the card number label as `cardNumLabelText`, and the text property of the error message label as `messageText`.
- ❑ We promote the text property of the PIN number text field as `pinFieldText`.
- ❑ We promote the `ActionPerformed` events of the **Ok** and **Cancel** buttons as `OkClicked` and `CancelClicked`.

SelectAccountPanel Class

The ATM application displays the `SelectAccountPanel` when the user's PIN is successfully validated. This panel is built from a `IList` bean that lists the account IDs related to the ATM card, a label associated with the `IList` bean, and two push buttons (Figure 89).

In the `SelectAccountPanel` the user selects an account from the list and switches to the `TransactionPanel` by clicking on the **Ok** button.

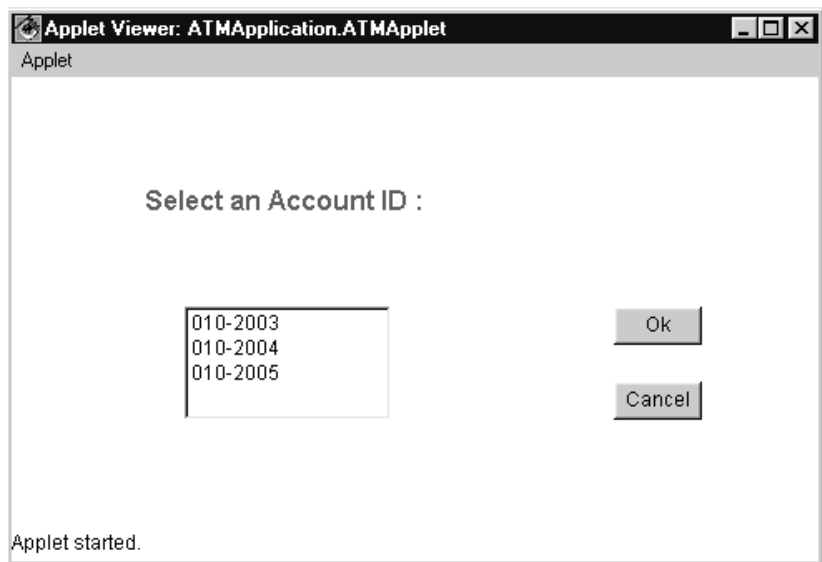


Figure 89. Layout of the `SelectAccountPanel`

Figure 90 shows the design of the panel inside the Visual Composition Editor.

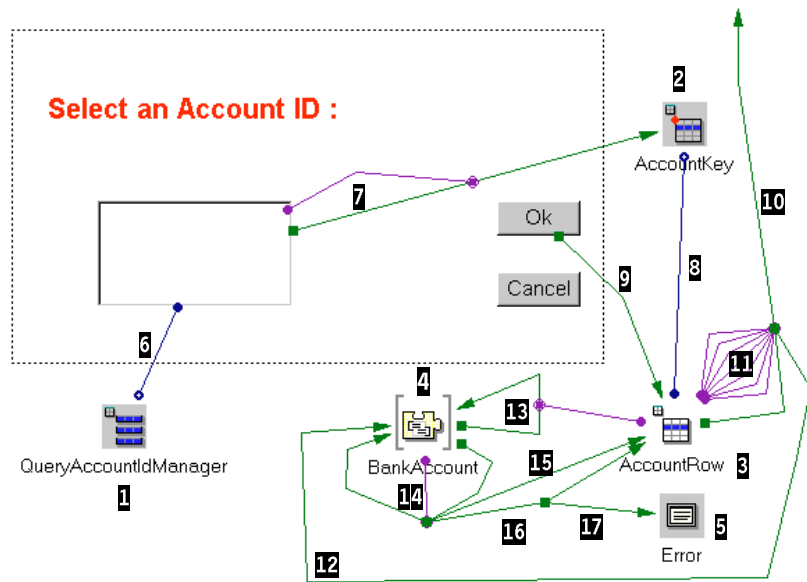


Figure 90. Visual Composition Editor: SelectAccountPanel

To complete the implementation of the panel we have to add the nonvisual beans that represent the business logic and the database, define an auxiliary method to construct the transaction object, connect the visual and nonvisual beans, and promote some of the features to make them accessible in the ATM application.

Nonvisual Beans

We add the following nonvisual beans to the free-form surface:

- ❑ A QueryaccidManager bean from the ATMDax package, called QueryAccountIdManager (1)
- ❑ An AccountDataId bean from the ATMDax package, called AccountKey (2)
- ❑ An Account bean from the ATMDax package, called Account-Row (3)
- ❑ A BankAccount bean variable, called BankAccount (4)
- ❑ A MessageBox bean, called Error (5)

Auxiliary Method

We add a method feature called `createAccount` to the `SelectAccountPanel`. This method creates and returns an instance of a `SavingAccount` or `CheckingAccount`, using the account data retrieved from the database:

```
public BankAccount createAccount (String accType, String accid,
                                String cardID, String custId,
                                java.math.BigDecimal bal,
                                java.math.BigDecimal min,
                                java.math.BigDecimal over) {
    if (accType.trim().equals("S"))
        return new SavingAccount(accType, accid, cardID, custId, bal, min);
    else
        return new CheckingAccount(accType, accid, cardID, custId, bal, over);
}
```

We will use this method to initialize the `BankAccount` variable in the free-form surface.

Connections

The following connections are necessary to implement the logic:

- ❑ `QueryAccountIdManager` has to populate the `IList` bean with the account IDs retrieved from the database. We connect the `items` property of `QueryAccountIdManager` with the `elements` property of the `IList` bean (6).
- ❑ When the user selects an account, we have to retrieve from the account table all the data related to the selected account. We connect the `itemStateChanged` event of the `IList` bean with the `accid` property of `AccountKey`, and we pass to this connection the `selectedItem` property of the `IList` bean as a parameter (7).
- ❑ The `AccountKey` bean is used to set the key for operations of the account bean. We connect the `this` property of `AccountKey` with the `dataId` property of `AccountRow` (8).
- ❑ When the **Ok** button is clicked we retrieve the selected account from the database. We connect the `actionPerformed` event of the **Ok** button with the `retrieve` method of `AccountRow` (9).
- ❑ When the retrieve is complete, `AccountRow` has all the information related to the account that the user has selected. From this information we create an instance of a `SavingAccount` or `CheckingAccount`, based on the account type retrieved. We connect the `retrieveComplete` event of `AccountRow` to the `createAccount` method of the `SelectAccountPanel` (10). We connect

the properties of AccountRow (acctype, accid, cardid, custid, balance, minamt, overdraft) as parameters to the previous connection (11).

- ❑ The createAccount method returns a BankAccount object that we use to set the BankAccount variable. We connect the normalResult of this createAccount connection to the this property of the BankAccount variable (12).
- ❑ The BankAccount object has to decode the account type code into the account type string. We connect the this event of the BankAccount variable to its DecodeAccountType method and pass the acctype property of AccountRow as a parameter (13).
- ❑ The BankAccount variable has to update the balance column of the ATM account table when the user performs a transaction and the account balance changes. Because the balance property of BankAccount is double and the database balance column is stored as BigDecimal, we have to convert the float value to BigDecimal:
 - We connect the balance event of the BankAccount variable to its converter method and pass the balance property as a parameter (14).
 - We connect the normalResult of the previous connection to the balance property of AccountRow (15) and to the update method of AccountRow (16).
 - We connect the exceptionOccurred feature of the last connection (update method) to the showException method of Error (17). We open the connection and click on the *Pass event data* check box.

Promote Features

We have to promote all the features of the AccountPanel that need to be accessed when the AccountPanel is embedded in the application. The select method of the QueryAccountIdManager and its parameter will be invoked from the application, and the BankAccount variable will be used in the TransactionPanel:

- ❑ We promote the select method and the param1 property of QueryAccountIdManager. To make it simple we do not modify the default names of the promoted features, that is, queryAccountIdManagerSelect and queryAccountIdManagerParam1.
- ❑ We promote the this property of the BankAccount variable. The system calls the promoted feature bankAccountThis.

TransactionPanel Class

The TransactionPanel is the last panel displayed by the ATM application (see Figure 91). It displays all information about the selected account and enables the user to perform withdrawal and deposit transactions and to view the transaction history related to the account.

To construct this panel we use the BorderLayout. In this way we can identify two different panel areas: a customer area in the North region of the BorderLayout, and an area for account transactions in the central region. Both areas contain a subordinate panel using the GridBagLayout. The customer area contains two labels; the transaction area contains 10 labels, one text field, three push buttons, and a choice (drop-down list). The layout is shown in Figure 91.

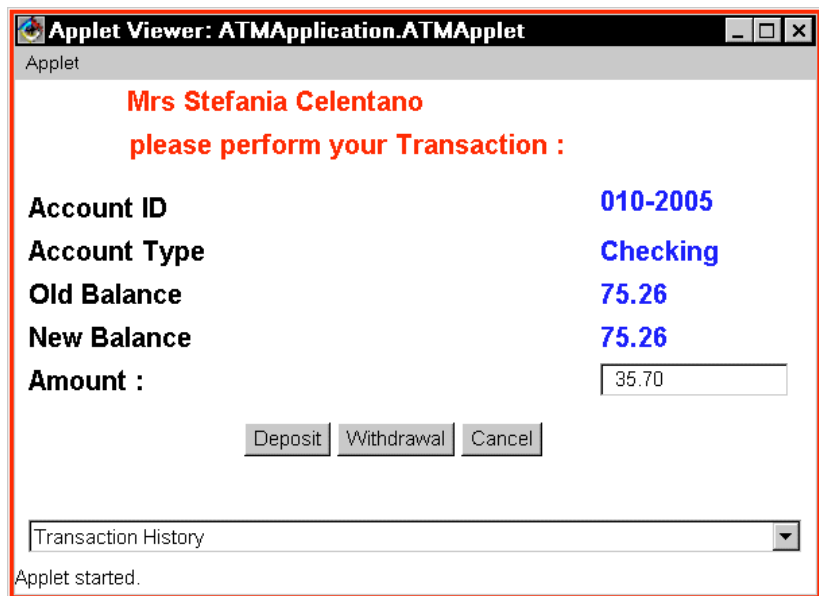


Figure 91. Layout of the TransactionPanel

We create the TransactionPanel, using the names Deposit, Withdrawal, and Cancel for the buttons, amount for the text field, TransactionList for the choice (history), and customerName, accountId, accountType, oldBalance, newBalance, and message for the labels with variable text data. The message label is used to inform the user when a withdrawal is not allowed.

Figure 92 shows the design of the panel inside the Visual Composition Editor.

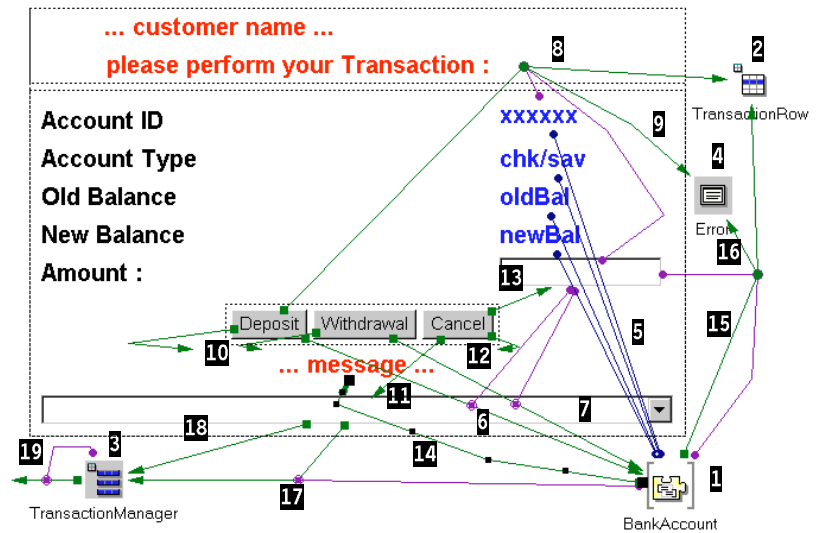


Figure 92. Visual Composition Editor: TransactionPanel

To complete the implementation of the panel we have to add the nonvisual beans that represent the business logic and the database, define an auxiliary method to construct the transaction object, connect the visual and nonvisual beans, and promote some of the features to make them accessible in the ATM application.

Nonvisual Beans

We add the following nonvisual beans to the free-form surface:

- ❑ A BankAccount variable, called BankAccount (1)
- ❑ A Transacc bean from the ATMDax package, called TransactionRow (2)
- ❑ A TransaccManager bean from the ATMDax package, called TransactionManager (3)
- ❑ A MessageBox bean, called Error (4)

Auxiliary Method

To refresh the transaction history list, we add a new method called `fillTransactionList` to the TransactionPanel. This method removes all previous items from the TransactionList choice and then adds to it all items selected from the database:

```

public void fillTransactionList(COM.ibm.ivj.javabeans.IVector arg1) {
    int numItem = arg1.size();
    getTransactionList().removeAll();
    getTransactionList().addItem("Transaction History");
    if (numItem != 0) {
        for (int i=0; i<numItem; i++)
            { getTransactionList().addItem(arg1.elementAt(i).toString());
        }
    }
    return;
}

```

Connections

The labels on the right side of the central region of the BorderLayout display the account data (account ID, account type, account old balance, and account new balance). The values are copied from the `BankAccount` variable (5):

- ❑ We connect the `accountId` property of the `BankAccount` variable with the text property of the `accountId` label.
- ❑ We connect the `typeDecod` property from the `BankAccount` variable with the text property of the `accountType` label.
- ❑ We connect the `oldBalance` property from the `BankAccount` variable with the text property of the `oldBalance` label.
- ❑ We connect the `balance` property from the `BankAccount` variable with the text property of the `newBalance` label.

The **Deposit** and **Withdrawal** buttons invoke the deposit and withdrawal transactions of the `BankAccount` variable, and they add the transactions to the transaction list:

- ❑ We connect the `actionPerformed` event of the **Deposit** button with the `deposit` method of the `BankAccount` variable and pass the text property of the amount text field as a parameter (6). We connect the **Withdrawal** button in the same way with the `withdraw` method (7).
- ❑ We connect the `actionPerformed` event of the **Deposit** button with the `addTransaction` method of `TransactionRow` (8). This connection requires three parameters. We pass to it the text property of the `accountId` label, the constant string “D,” and the text property of the amount text field.
- ❑ We connect the `exceptionOccurred` feature of the previous connection with the `showException` method of `Error` (9) and pass the event data (open the connection and select the check box).
- ❑ We connect the `actionPerformed` event of the **Deposit** button with the text property of the message label and set the text to blank (10). We create the same connection from the **Withdrawal** button.

The **Cancel** button closes the panel and switches to the account panel, but before that it has to remove all transactions from the `TransactionList` and clean the amount text field and the message label:

- ❑ We connect the `actionPerformed` event of the **Cancel** button to the `removeAll` method of `TransactionList` (11).
- ❑ We connect the `actionPerformed` event of **Cancel** to the `text` property of the amount text field bean and set the text to blank (12).
- ❑ We connect the `actionPerformed` event of **Cancel** to the `text` property of the message label and set the text to blank (13).

The `BankAccount` variable has to indicate the result of the withdrawal transaction by signaling the `withdrawalDone` or `withdrawalFailed` event. In the case of the `withdrawalFailed` event, the message label displays a message to the user, otherwise the withdrawal transaction is added to the transaction list:

- ❑ We connect the `withdrawalFailed` event of the `BankAccount` variable with the `text` property of the message label and set the text to the fixed string “You have insufficient funds” (14).
- ❑ We connect the `withdrawalDone` event of the `BankAccount` variable with the `addTransaction` method of `TransactionRow` (15). This connection requires three parameters. Pass to it the `accountId` property, the fixed String “W,” and the `text` property of the amount text field.
- ❑ We connect the `exceptionOccurred` feature of the previous connection with `showException` method of `Error` and pass the event data (16).

When the user selects the transaction history (choice drop-down), the `TransactionList` displays all the transactions related to the selected account. The `TransactionManager` selects these transactions from the ATM trans table:

- ❑ We connect the `mousePressed` event of `TransactionList` with the `param1` property of the `TransactionManager` bean and pass the `accountId` property of the `BankAccount` (17).
- ❑ We connect the `mouseReleased` event of `TransactionList` with the `select` method of `TransactionManager` and complete the connection with a blank parameter value (18).
- ❑ We connect the `selectComplete` event of the `TransactionManager` with the `fillTransactionList` method of the `TransactionPanel` and pass the `items` property as parameter (19).

Promote Features

In the transaction panel the `BankAccount` variable is the account selected by the user and the `customerName` label is the full name of the customer. Therefore, we have to set these objects before the `TransactionPanel` is displayed:

- ❑ We promote the `this` property of the `BankAccount` variable and call the promoted feature `bankAccountThis`.
- ❑ We promote the `text` property of the `customerName` label and call the promoted feature `customerNameText`.

In the `TransactionPanel` we use `TransactionList` to display the transaction history. When the ATM application displays the panel, we want to display the text “Transaction History,” so that the user understands the function of this object.

- ❑ We promote the `add(java.lang.String)` method of the `TransactionList` and call the feature `transactionListAdd(java.lang.String)`.
- ❑ We promote the `actionPerformed` event of the **Cancel** button and call the feature `CancelClicked`.

Application Flow

In this section we explain how to create the ATM applet by assembling the four panels built in the previous section.

The ATM applet is a subclass of the `Panel` class. We use a `CardLayout` manager that enables the user to switch from one panel to the next by using the push buttons available in each panel.

The design of the applet requires the following steps:

- ❑ Applet layout
- ❑ Panel switching
- ❑ Sharing the `Card` object
- ❑ Database connection

Applet Layout

To build the applet we assemble the four panels in a card layout:

- ❑ We create an applet with the name `ATMApplet` in the `ATMAp-
plication` package.
- ❑ We change the layout property of the `ATMApplet` to `CardLay-
out` and we call the applet panel `MainPanel`. This panel con-
tains the four panels of the ATM application.
- ❑ We add to the `MainPanel` the four panels we built, that is,
`CardPanel`, `PinPanel`, `SelectAccountPanel`, and `Transaction-
Panel`. We call these beans `CardPanel`, `PinPanel`, `SelectAc-
countPanel`, and `TransactionPanel`.
- ❑ We add the following nonvisual beans to the free-form surface:
 - An `AccountDatastore` bean from the `ATMDax` package,
called `Connection` (1)
 - A `Card` variable, called `Card` (2)
 - A `CardLayout` variable, called `CardManager` (3)
 - A `MessageBox` bean, called `Error` (4)

Figure 93 shows the connection diagram for the main panel, and Figure 94 shows the Beans List of the applet.

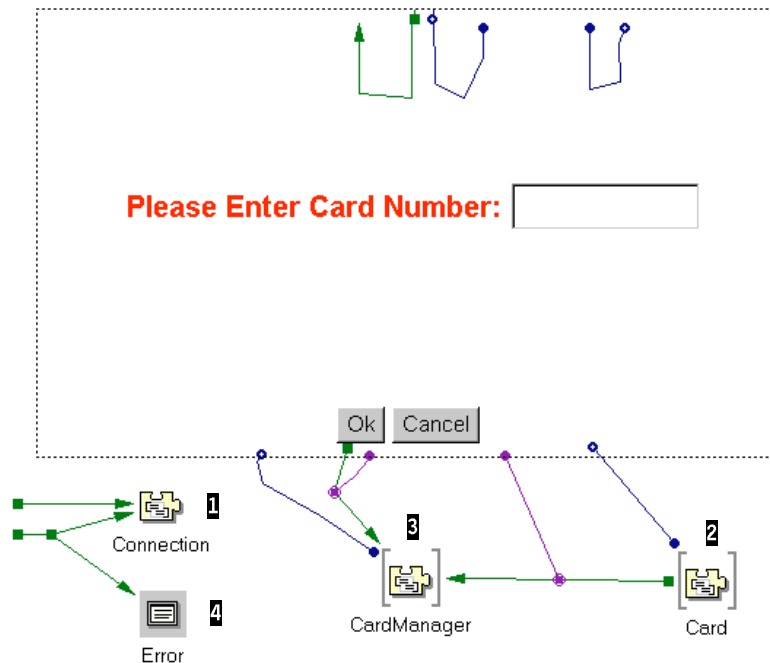


Figure 93. ATM Applet Main Panel and Beans

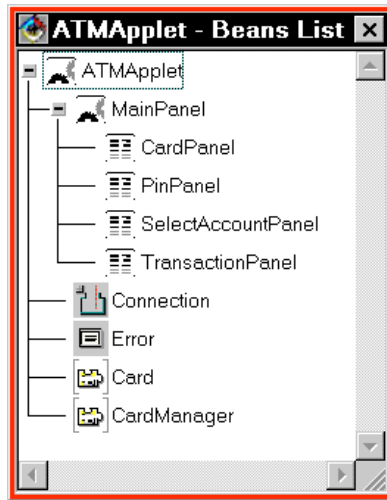


Figure 94. ATM Applet Beans List

Panel Switching

The next step is to implement the switching from panel to panel in the card layout of the applet. Use the Beans List (Figure 94) to select one of the four panels to make additional connections.

Connecting the CardPanel

The CardPanel sets the Card variable and switches to the PinPanel:

- ❑ We connect the card property of the CardPanel with the this property of the Card variable. In this way, we transmit the Card object instantiated by the CardPanel to all other panels through the Card variable.
- ❑ We connect the customerInfo property of the CardPanel with the customerNameLabelText property of the PinPanel to propagate the full name of the customer to the PinPanel. When we create this property-to-property connection, the system sets the source event to the customerInfo event and the target event to <none>. We change the target event to the CancelClicked event to make the connection bidirectional, that is, the customerInfo event of the CardPanel sets the customerNameLabelText property of the PinPanel, and the CancelClicked event of the PinPanel sets the customerInfo property of the CardPanel with the customerNameLabelText property value.
- ❑ We connect the customerInfo property of the CardPanel with the customerNameText property of the TransactionPanel.

- ❑ We connect the `OkClicked` event of the `CardPanel` with the `cardIdFieldText` property of the `CardPanel`. We do not set the parameter of the connection, and we leave the connection dotted, that is, the `cardIdFieldText` is set to an empty string.
- ❑ We connect the `OkClicked` event of the `CardPanel` with the next method of the `CardManager` variable. We connect the parent parameter of this connection to the `this` property of the `MainPanel`.

After these connections have been completed, the `CardPanel` Reorder connections window should look like that shown in Figure 95.

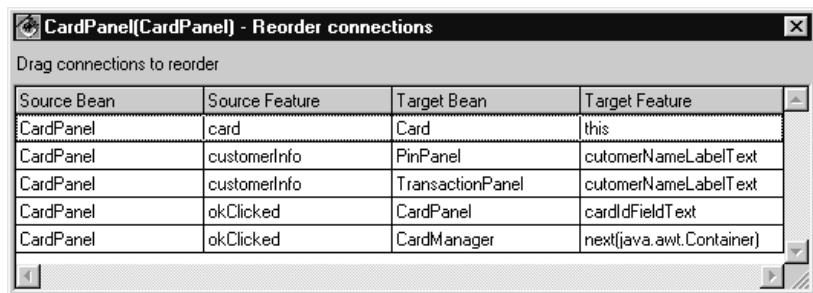


Figure 95. CardPanel Reorder Connections Window

Connecting the PinPanel

From the `CardPanel` the user switches to the `PinPanel`, where the full name and the card number are displayed. The user then enters the PIN to proceed to the `SelectAccountPanel`:

- ❑ We connect the `cardNumber` property of the `Card` variable with the `cardNumberLabelText` property of the `PinPanel` to display the card number entered in the `CardPanel`.
- ❑ We connect the `CancelClicked` event of the `PinPanel` with the first method of the `CardManager` variable and connect the parent parameter of this connection to the `this` property of the `MainPanel`.
- ❑ We connect the `OkClicked` event of the `PinPanel` with the `checkPin` method of the `Card` variable and pass the `pinFieldText` property of the `PinPanel` as a parameter.

Notice that we do not make a connection to show the customer name on the top of this panel because we already implemented this behavior in the `CardPanel`.

Figure 96 shows the Reorder connections window of the `PinPanel`.

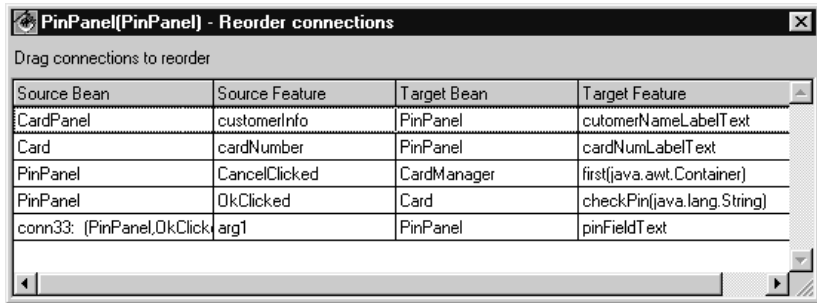


Figure 96. PinPanel Reorder Connections Window

Connecting the SelectAccountPanel

When the ATM application displays the SelectAccountPanel, the user selects an account ID and clicks on the **Ok** button to switch to the TransactionPanel to see the account data and to perform transactions. The user can also choose the **Cancel** button to return to the CardPanel.

To implement this behavior, the SelectAccountPanel has to create an instance of a BankAccount object (a SavingAccount or a CheckingAccount) after the user has selected an account ID. Then the SelectAccountPanel has to forward this BankAccount object to the TransactionPanel:

- ❑ We connect the CancelClicked event of the SelectAccountPanel with the first method of the CardManager variable and connect the parent parameter of this connection to the this property of the MainPanel.
- ❑ We connect the OkClicked event of the SelectAccountPanel with the last (or next) method of the CardManager variable and connect the parent parameter of this connection to the this property of the MainPanel.
- ❑ We connect the OkClicked event of the SelectAccountPanel with the transactionListAdd method of the TransactionPanel. We set the parameter as the fixed string “Transaction History” to display an initial string in the TransactionList choice bean.
- ❑ We connect the bankAccountThis property of the SelectAccountPanel with the bankAccountThis property of the TransactionPanel to forward the BankAccount object from the SelectAccountPanel to the TransactionPanel.

Figure 97 shows the Reorder connections window of the SelectAccountPanel.

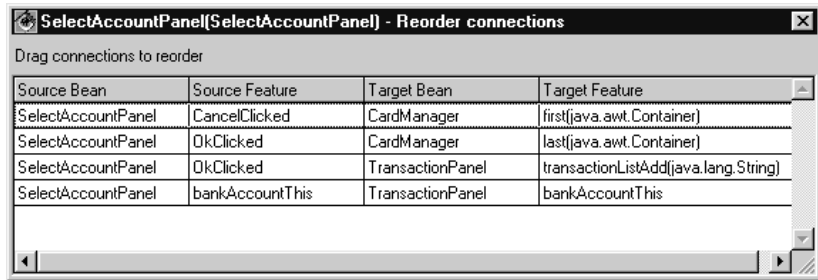


Figure 97. SelectAccountPanel Reorder Connections Window

Connecting the TransactionPanel

The TransactionPanel displays the customer name and all data related to the selected account (account ID, account type, account balance). From this panel the user can switch to the previous panel to select another account ID.

In this panel, the customer label at the top is initialized from the customerInfo property of the CardPanel, and the account information is initialized from the BankAccount variable. The BankAccount variable is already connected from the SelectAccountPanel. We only have to implement the return switch to the SelectAccountPanel:

- ❑ We connect the CancelClicked event of the TransactionPanel to the previous method of the CardManager variable and connect the parent parameter of this connection to the this feature of the MainPanel.

Figure 98 shows the Reorder connections Window of the TransactionPanel.

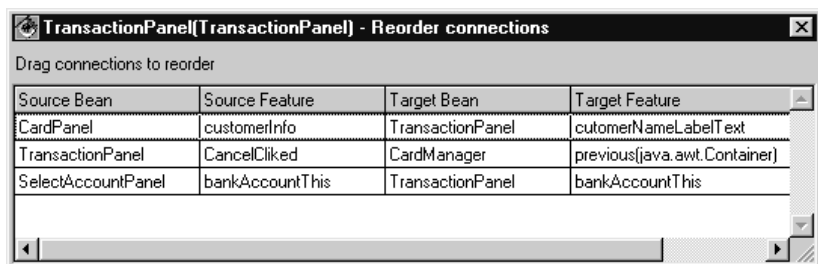


Figure 98. TransactionPanel Reorder Connections Window

Sharing the Card Object

The Card object, represented by the Card variable, is instantiated in the CardPanel and has information that the other panels need. The PinPanel displays the card number to the user, and the SelectAccountPanel need the card number to select and display all the account IDs. In addition, the Card object contains the checkPin method that fires the pinCheckedOk or pinCheckedNotOk event.

When the card **variable** validates the PIN successfully, it switches the ATM application to the SelectAccountPanel and selects all the account IDs related to the card number:

- ❑ We connect the cardNumber property of the Card variable with the queryAccountIdManagerParam1 property of the SelectAccountPanel.
- ❑ We connect the pinCheckedOk event of the Card variable with the queryAccountIdManagerSelect method of the SelectAccountPanel.
- ❑ We connect the pinCheckedOk event of the Card variable with the next method of the CardManager variable and connect the parent parameter of this connection to the this feature of the MainPanel.

Before switching from the PinPanel to the SelectAccountPanel, we have to initialize the pinFieldText feature of the PinPanel for data entry:

- ❑ We connect the pinCheckedOk event of the Card variable to pinFieldText property of the PinPanel and set the parameter to an empty string.

If the PIN validation is not successful, the ATM application displays an error message to the user and resets the PIN entry field:

- ❑ We connect the pinCheckedNotOk event of the Card variable with the messageText property of the PinPanel and pass the constant string “Pin not valid. Please, try again.”
- ❑ We connect the pinCheckedNotOk event of the Card variable with the pinFieldText property of the PinPanel and set the parameter to an empty string.

Figure 99 shows the Reorder connections window for the Card variable.

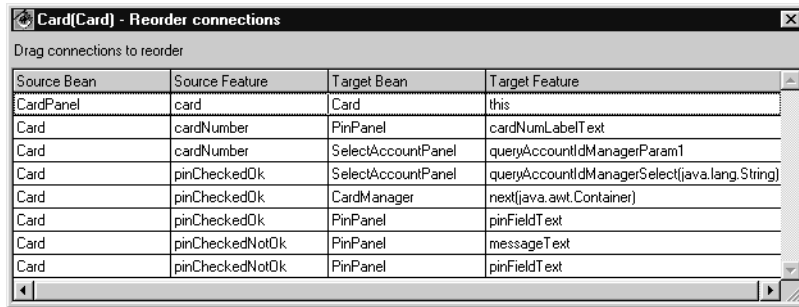


Figure 99. Card Variable Reorder Connections Window

Database Connection

To complete the ATM applet, we have to connect to and disconnect from the database. The database connection is the first operation that the applet has to perform when it starts, and the database disconnection is the last operation before the applet is destroyed. An error message is displayed if an error occurs during the database connection.

- ☐ We connect the `init` event of the `ATMApplet` with the `connect` method of `Connection` (the data store bean) and pass the user ID and password as parameters.
- ☐ We connect the `exceptionOccurred` event of this connection with the `showException` method of `Error` and pass the event data (open the connection and select the check box).
- ☐ We connect the `destroy` event of the `ATMApplet` with the `disconnect` method of `Connection`.

Running the ATM Application

Now we are ready to run the applet and play with the sample data. Hopefully it will work!

6

Remote Method Invocation and RMI Access Builder

RMI is the feature in Java that enables you to create distributed applications. In this chapter we explain what RMI is and how it is supported by VisualAge for Java. After we look at the big picture, we go into the details of its implementation in Java and build a simple RMI application.

In the second part of the chapter, we explain how VisualAge for Java Enterprise supports the use of RMI with the RMI Access Builder. We explain the concept of the RMI Access Builder and create a simple bank account applet.

Overview

In “Database Application Architectures” on page 34 we introduced two-tier and three-tier architectures, and, using JDBC, we have created applications that can access either a local or a remote database. First, we used a stand-alone or two-tier architecture as shown in Figure 100.

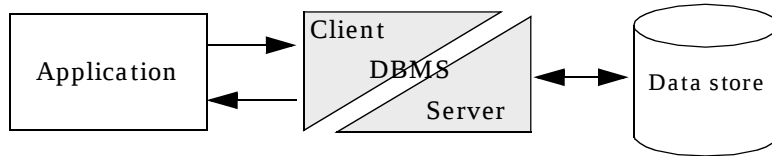


Figure 100. Two-Tier Architecture

A two-tier architecture requires that all functions are performed and all data is accessed on the client, which then becomes a so-called fat client. If we want to deploy an application over the Internet or intranet, fat clients are not exactly what we are looking for.

One approach of shifting some processing to a server is to use a three-tier architecture and the JDBC generic network protocol driver (see page 43). All database requests are routed from the client (applet) to a JDBC server connected to the DBMS client. In such a design, all SQL is executed in the DBMS client and server. The client application has to set up SQL statements and process the results of SQL execution. All SQL data is communicated to the DBMS client (Figure 101).

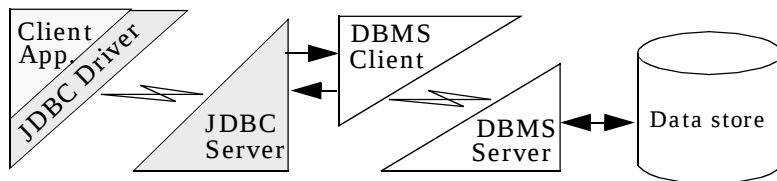


Figure 101. Three-Tier Architecture with JDBC Network Protocol Driver

To minimize the size of the applet, reduce network communication, and shift all SQL setup and processing to the second (middle) tier, we need a different approach. We want to send tiny applets over the network, instead of full-blown applications labeled as applets.

We have to divide the application into a thin client and a server part. The server part processes all data access and sends only consolidated result objects to the thin client. The client handles the presentation (GUI), validates input data, and communicates requests to the application server.

Such a design makes it possible to spread the application over multiple platforms, such as desktop PCs, departmental servers, and enterprise host systems (Figure 102).

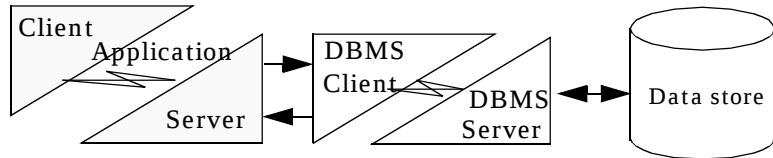


Figure 102. Three-Tier Architecture with Client/Server Application

Having said that, we have to think about how the thin client is going to talk to its server counterpart. One implementation is RMI, a Java-based client/server protocol.

Using RMI for Distributed Processing

To create object-oriented, distributed client/server applications, the different parts of your application that run on different computers and platforms must communicate with each other. This is where RMI comes into play.

Basically you have three ways of implementing distributed processing:

1. Create your own communication interface to some underlying protocol (APPC, TCP/IP).
2. Buy an Object Request Broker (ORB).
3. Use Java's RMI.

You might consider creating your own communication interface if you are thinking of a rather small application, with clearly defined and limited communication needs between the different parts of the application. You also might consider that solution, if avoiding any type of overhead is an issue. However, you will have to put a lot of effort into a nonapplication area to build the interface, and any changes (application, platform, or communication) will require major effort to keep the application running.

Buying an ORB would work if you use different programming languages, on different platforms, or if you are building an enterprise-wide solution.

RMI enables you to implement communication between remote objects in a pure Java environment. As you will see, Java provides all the necessary classes and interfaces to send messages from one object to another, even if these objects run in different computers.

How Does RMI Work?

From a conceptual point of view, RMI enables a Java application to call a remote Java object as if it were on the local machine. While, from an application point of view, one object invokes a method in another object, RMI provides the necessary services to locate the remote object, route the method call with all its parameters through TCP/IP to the server, invoke the method on the server object, and pass back the return object along the same path (Figure 103).

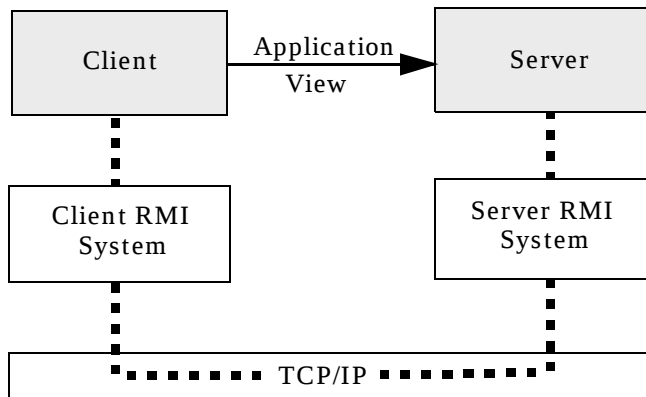


Figure 103. RMI Conceptual View

Squeezing Objects through a Network

RMI sends objects over the network. As TCP/IP is not aware of objects and knows only bits and bytes, objects have to be converted into a stream of bytes. This is not too difficult as long as these objects are simple data types, such as numbers or strings. In fact, simple data types have been transmitted before in a feature known as *remote procedure call* (RPC).

RPC lets you call programs on remote computers and transforms parameters and return values into bytes and vice versa. In an object-oriented environment, however, things get a little more complicated. We want to send objects, which may have properties that reference yet more objects, ending in a graph of referenced objects. We call converting such a graph of objects into a stream of bytes and restoring it at the target location, respectively, *serialization* and *deserialization*. If an object (or a graph of objects) is written to or read from a buffer, this process of serialization and deserialization is often referred to as *marshaling* and *demarshaling*.

RMI enables the marshaling and demarshaling of objects. However, each object that is meant to use these services must ensure that its state information can be transformed into a byte stream. Therefore, the definition of the object's class or of one of its super-classes has to include the *java.io.Serializable* interface. This interface has no methods defined; it just indicates that the objects of the class can be handled by the *readObject()* and *writeObject()* methods of the I/O stream classes. If a class implements serialization, all values of the instance variable of that class have to be serializable. Therefore if an instance variable points to another object, the class of that object has to be serializable too. Java's basic data types are serializable, as are all classes representing basic data types (for example, *java.lang.Integer*), and arrays and vectors.

If you want to exclude an instance variable from serialization, you mark it with the *transient* keyword. A variable marked as *transient* will not be written to the output stream, and on deserialization a value is not assigned to the variable. In the context of RMI this might be useful if you want an object to be passed as a parameter but do not need or want to pass the entire graph of referenced objects over the network.

RMI Architecture

RMI calls are handled by different layers until they eventually are communicated from the client to the server, where they cross the same number of layers until they arrive at the server object.

Figure 104 illustrates the RMI architecture.

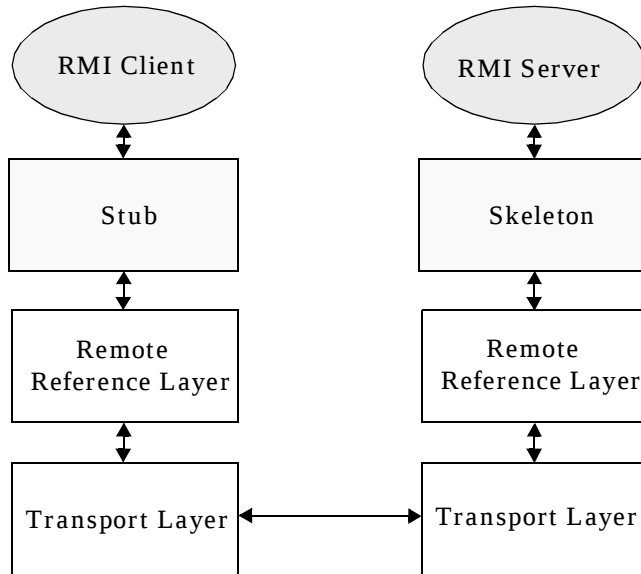


Figure 104. RMI Architecture

Stub and Skeleton

The stub and skeleton classes are the interfaces between the application layer and the rest of the RMI system. The stub resides on the client and marshals the arguments, triggers the call to the remote object by calling the remote reference layer (RRL), and unmarshals return objects and exceptions.

The skeleton on the server side is responsible for unmarshaling the arguments, calling the server object, and marshaling the return objects and the exceptions.

Remote Reference Layer

The RRL has a client and a server component. It is responsible for maintaining a reference protocol between the components that is independent of a specific stub or skeleton. It keeps references and reconnects if a connection is lost.

Transport Layer

The transport layer creates and monitors the connections on behalf of the RRL. It establishes socket connections and passes the connections to the RRL. It also listens for incoming calls and sets up connections for them.

Tools

JDK 1.1 includes two RMI tools, the RMI compiler and the RMI registry.

RMI Compiler

The RMI compiler is used after the classes and interfaces to be distributed are compiled. It generates the stub and skeleton classes you need for communication. To invoke the RMI compiler, use the *rmic* command with the fully qualified class or interface name as a parameter. For example, if you have a class named *MyRmiObject* in the *myRmiPackage* package, the following command generates *MyRmiObject_Stub.class* and *MyRmiObject_Skel.class*:

```
rmic myRmiPackage.MyRmiObject
```

The stub and skeleton classes are then used to finish the application on both the client and server side.

RMI Registry

The RMI registry is a program that provides naming lookup services at run time. It must be running before an RMI server object can be instantiated. A server object has to register itself with the RMI registry to be accessible from clients.

To start the registry program, use this command:

```
start rmiregistry          (Windows)
start rmiregst             (OS/2)
```

RMI Development Process

The development process for an RMI application, including registration of the server and lookup in the client, are best explained with a simple example.

The development process consists of the following steps (Figure 105):

- ❑ Define the public interface of the server (I).

The public interface of the server defines the methods that can be invoked from the client through RMI.

- ❑ Write the server implementation (2).

The server class implements the methods defined in the public interface of the server. In addition, the server class uses special RMI coding to set up a security manager and to register the server with the RMI registry.

- ❑ Compile the server interface and the server classes with the Java compiler (3).
- ❑ Run the RMI compiler on the compiled server class to create stub and skeleton classes(4).
- ❑ Write and compile the client logic (5).

The client logic uses special RMI coding to look up the server by name and get a reference to a local proxy object (the stub) that represents the server object. It can then use the public server methods through the local proxy object.

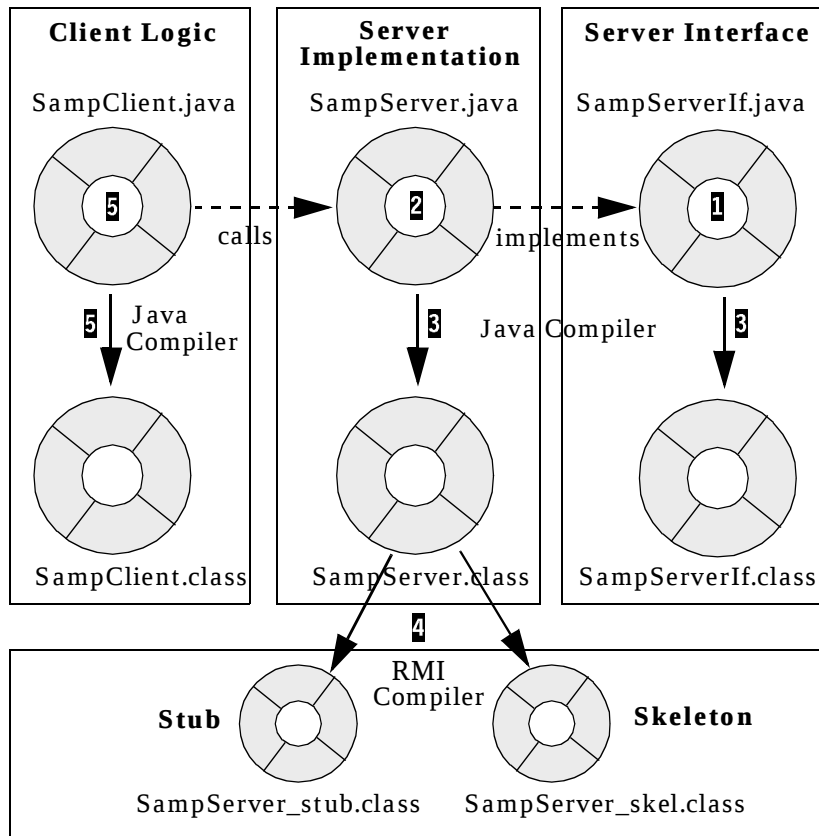


Figure 105. RMI Development Process

Special Coding

Both server and client classes require some special code to make the RMI connection work:

- ❑ The server interface definition must be a subclass of `java.rmi.Remote`:

```
public interface ServerInterface extends Remote
```

- ❑ Every public server method can throw the `java.rmi.RemoteException`:

```
public returnType methodName(...parms...) throws RemoteException
```

- ❑ The server class must be a subclass of `java.rmi.server.RemoteServer`; typically a subclass of `java.rmi.server.UnicastRemoteObject` is used:

```
public class ServerClass extends UnicastRemoteObject
    implements ServerInterface
```

- ❑ The server class must set up a security manager:

```
System.setSecurityManager(new RMISecurityManager());
```

- ❑ The server class must register with the RMI registry:

```
serverInstance = new ServerClass(); // constructor
Naming.bind("rmi://ServerName", (Remote)serverInstance);
```

The server registers under a user-defined name, *ServerName*, with the `java.rmi.Naming` class.

- ❑ The client class must set up a security manager:

```
System.setSecurityManager(new RMISecurityManager());
```

- ❑ The client class must look up the server on the remote machine to get a reference to a proxy object for invoking the remote methods:

```
urlstring = "rmi://serverHostname/ServerName"
serverObject=(ServerClassInterface)Naming.lookup(urlstring);
```

The client must know the TCP/IP host name of the server machine, *serverHostname*, and the name under which the server registered, *ServerName*. After calling the `lookup` method of the `java.rmi.Naming` class, the client can invoke the public methods of the server, using the server proxy object returned:

```
value = serverObject.methodName(...parms...)
```

Execution Environment

Figure 106 shows the RMI execution environment and the sequence of operations.

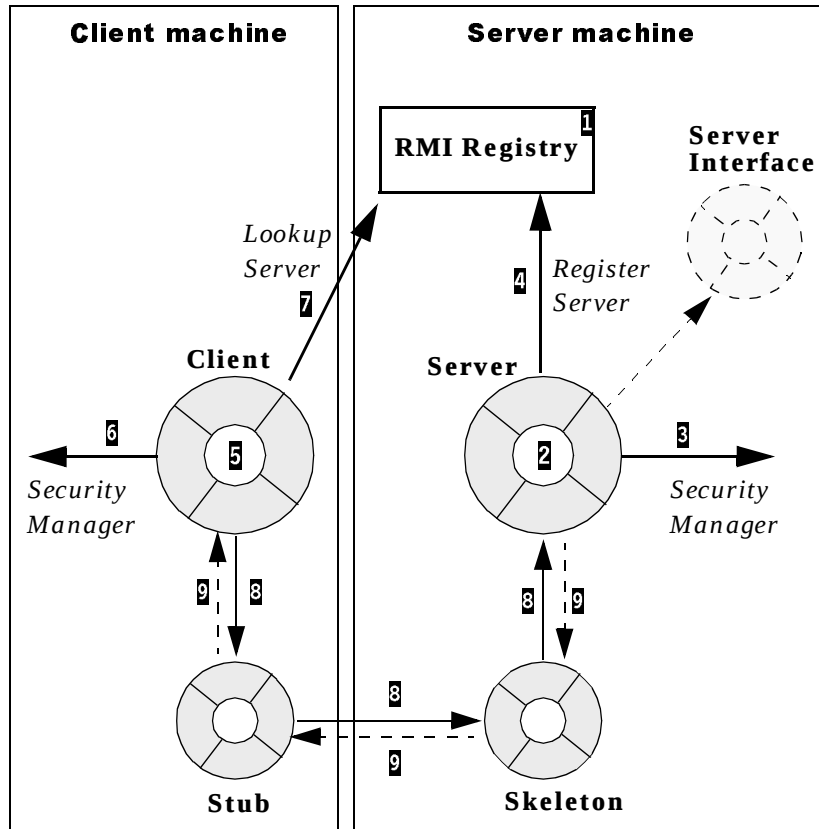


Figure 106. RMI Execution Environment

The sequence of operations to run an RMI application is:

- ❑ Start the RMI registry on the server machine (1).
- ❑ Start the server application (2), which sets up a security manager (3) and registers itself with the RMI registry (4).
- ❑ Start the client application or applet (5), which sets up a security manager (6) and uses the RMI registry on the remote machine to look up the server (7).
- ❑ The client can now use a local object to invoke methods through the stub and skeleton on the server object (8). Result objects of such calls are routed back to the client object (9).

Native RMI Example

To review the RMI development process and execution environment, and to understand more about RMI coding, let us look at a simple, native RMI application.

We create an RMI server that concatenates an input string with its reverse string and returns the concatenated string to the caller. We use the Java package `RMINative` to implement this example.

Public Interface of the Server

The public interface of the server defines the methods that can be invoked from the client. For our example the interface defines the `reverseAppend` method:

```
/* RMINative\SampServerIf.java */
package RMINative;
import java.rmi.*;
public interface SampServerIf extends Remote
{
    public abstract String reverseAppend(String str)
                                throws RemoteException;
}
```

Every public method can throw the `RemoteException` if the communication breaks down, for example.

Server Implementation

Let us look at the server code in small pieces. The server must be a subclass of a remote object and has to implement the public interface:

```
/* RMINative\SampServer.java */
package RMINative;
import java.rmi.*;
import java.rmi.server.*;
public class SampServer extends UnicastRemoteObject
    implements RMINative.SampServerIf
{
}
```

The main method has to set up a security manager, instantiate the server object with the default constructor, and register the server object with the registry under a name (`Reverser`) that will be used by the client:

```
public static void main(String args[])
{
    RMINative.SampServer reverserObject = null;
    try
    {
        System.out.println("Setting up the Security Manager ...");
```

```
        System.setSecurityManager(new RMISecurityManager());
        System.out.println("Publishing the \"Reverser\" object:
                           rmi:///Reverser");
        reverserObject = new RMINative.SampServer();
        Naming.rebind("rmi:///Reverser", (Remote)reverserObject);
        System.out.println("Reverser server is ready ...");
    }
    catch (Exception e)
    {
        System.out.println("Exception "+e+" caught: \n"+e.getMessage());
    }
    return;
}
```

The constructor allocates the server object:

```
public SampServer() throws RemoteException
{
    super();
    return;
}
```

The reverseAppend method is called by the client through the stub and the skeleton. It can use other methods to perform the work:

```
public String reverseAppend(String str) throws RemoteException
{
    String result = null;
    try
    {
        System.out.println("Client "+getClientHost()+" says: "+str);
        result = performWork(str);
        return result;
    }
    catch (Exception e)
    {
        System.out.println("Exception "+e+" caught: \n"+e.getMessage());
        return "Error";
    }
}
```

The performWork method is a private method that does the actual job of constructing the result string:

```
private String performWork(String in)
{
    StringBuffer buf;
    buf = (new StringBuffer(in)).reverse();
    return in+"-"+(new String(buf));
}
// end of class
```

Note how an RMI server is registered:

```
Naming.rebind("rmi:///Reverser", (Remote)reverserObject);
```


The general format of an RMI server URL is:

```
rmi://hostname/ServerName
```

where *hostname* is the TCP/IP host name or address of the server machine. This format will be used in the client to find the server object.

Stub and Skeleton

After compiling both the public interface of the server and the server implementation, we run the RMI compiler to create the stub and skeleton classes:

```
javac SampServerIf.java
javac SampServer.java
rmic RMINative.SampServer
```

Note that we must give the complete qualified name of the server class. The RMI compiler creates:

```
SampServer_Stub.class
SampServer_Skel.class
```

Client Logic

Now we can write and compile the client. The client class has no special coding:

```
/* RMINative\SampClient.java */
package RMINative;
import java.rmi.*;
import java.io.*;
public class SampClient
{
```

In the main method we prepare all the local variables. The TCP/IP host name of the server defaults to an empty string for the local system, or it can be given as the only parameter when starting the client:

```
    public static void main(String args[])
    {
        RMINative.SampServerIf reverserObject = null;
        BufferedReader infile = new BufferedReader(
            new InputStreamReader(System.in));

        String hostname = "";
        String urlstring;
        String userInput;
        String result;
        if (args.length == 1) { hostname = args[0]; };
    }
```

We construct the RMI server URL, using the host name of the server machine and the known server name, register a security manager, and look up the client from the remote registry:

```
urlstring = "rmi://" + hostname + "/Reverser";
if (hostname == "") hostname = "{local}"; // for the message
try
{
    System.out.println("Registering the Security Manager ...");
    System.setSecurityManager( new RMISecurityManager() );
    System.out.println("Looking up the Reverser on "+hostname+
        "... Please Wait !");
    reverserObject= (RMINative.SampServerIf)Naming.lookup
        (urlstring);
}
```

The lookup method creates a local proxy object that represents the server. We can invoke the public methods on the server object as if the server were on the local client machine:

```
do {
    System.out.println("What should I say to the server ?
        (or type: end)");
    userInput = infile.readLine();
    System.out.println(" - calling server: "+userInput);
    result = reverserObject.reverseAppend(userInput);
    System.out.println(" - server replied: "+result);
    } while (!userInput.equals("end"));
}
```

Exceptions must be handled in case of communication, URL, or I/O problems:

```
catch (RemoteException e1)
{System.out.println("Something is wrong with the RMI connection!");
    System.out.println("Exception "+e1+" caught: \n"+e1.getMessage());}
catch (java.net.MalformedURLException e2)
{System.out.println("The URL is not valid: "+urlstring);
    System.out.println("Exception "+e2+" caught: \n"+e2.getMessage());}
catch (Exception e3)
{System.out.println("Exception "+e3+" caught: \n"+e3.getMessage());}

System.out.println("End of RMI Client\n");
return;
}
```

The constructor allocates an object of the client class:

```
private SampClient() { return; } // constructor

} // end of class
```

Run the RMI Application

To run the native RMI application on a single machine, make sure that:

- ❑ The `RMINative` package directory is a subdirectory in the `CLASSPATH`
- ❑ The TCP/IP loopback interface has been configured

Start the registry process and the server and start the client when the server is ready:

```
start rmiregistry           (rmiregst on OS/2)
start java SampServer
pause                       (wait for the server ready message)
java SampClient
```

To run the native RMI application in a client/server configuration, make sure that all the code is available on both systems in a subdirectory of `CLASSPATH`. Start the registry and the server application on the server machine and, when ready, start the client application with the host name of the server as a parameter:

```
server:  start rmiregistry           (rmiregst on OS/2)
         start java SampServer

client:  java SampClient serverhostname
```

Stop the RMI Application

Stop the client program by entering *end* as user input. Stop the server and the registry processes, using the *Ctrl-C* key combination in their respective windows.

More on Native RMI

For a detailed description of how to create native Java RMI applications, visit the following Web site:

<http://java.sun.com/products/jdk/1.1/docs/guide/rmi>

RMI with VisualAge for Java

Using RMI to create a distributed application involves a number of administrative tasks and tends to pollute application code with RMI-specific code that is scattered over the application. Another drawback of RMI is its lack of support of events. The VisualAge for Java Enterprise RMI Access Builder solves these problems. It lets you take advantage of RMI without forcing you to care too much about what is going on under the covers.

RMI Access Builder

The RMI Access Builder generates distribution code from an existing server class (usually a Java bean) to support the distribution of your application using RMI. The generated code includes a client-side proxy bean that can be used in the Visual Composition Editor as if the server were a local object.

The RMI Access Builder has the following advantages:

- ❑ A SmartGuide leads you through the process of creating all classes and interfaces needed to distribute your application.
- ❑ Your existing server class is not changed during this process and is not aware of any aspect of the distribution. All definitions and tasks related to distribution are in a generated RMI server class that wraps the server bean.
- ❑ Event services, which are not available in native RMI, are available in the RMI Access Builder. An event fired by the client object is propagated to the server, and an event fired on the server is passed to the client.
- ❑ A Remote Object Instance Manager on the server provides a GUI for administrative tasks of managing the server objects.

You find the classes providing the RMI Access Builder support in the *COM.ibm.ivj.eab.rmi.client* and *COM.ibm.ivj.eab.rmi.server* packages.

Development Process

The development process is quite simplified with the RMI Access Builder because many of the required classes and interfaces are generated automatically (Figure 107).

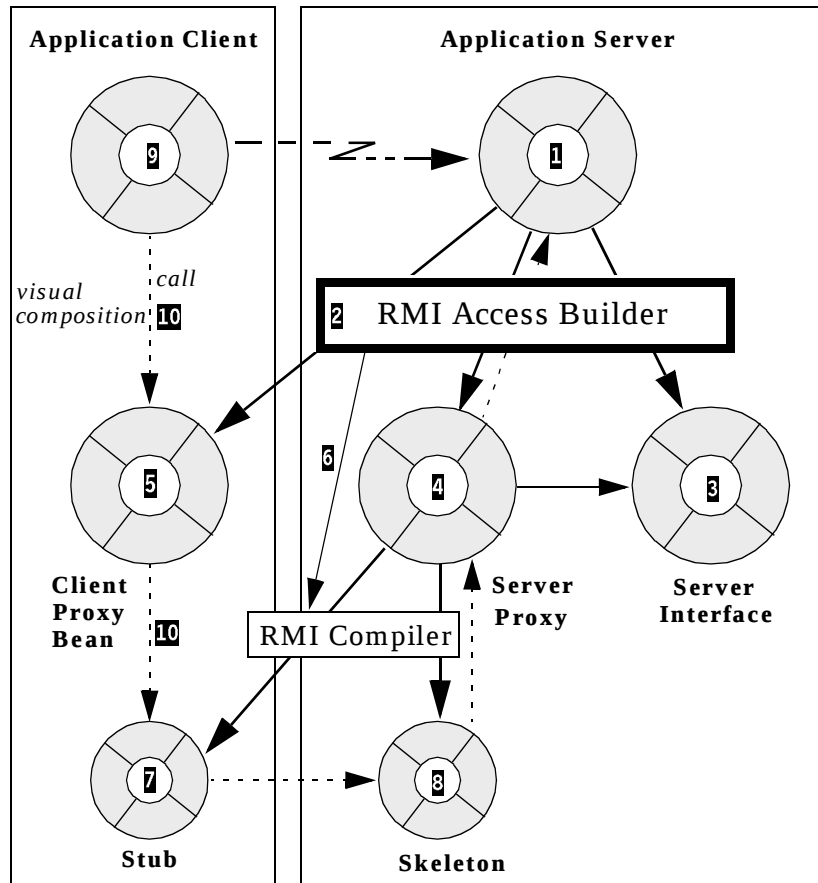


Figure 107. Development Process with RMI Access Builder

- ❑ The application server (1) is developed without any special RMI coding. All public methods are part of the RMI public interface.
- ❑ The RMI Access Builder (2) generates the RMI server interface (3) from the public methods of the application server, the server proxy (4), and the client-side proxy bean (5) that represents the application server.
- ❑ The RMI Access Builder calls the RMI compiler (6) that generates the stub (7) and the skeleton (8) classes.
- ❑ The client application (9) is developed through the client-side proxy (5) without any special RMI coding.
- ❑ The client application calls the methods of the client proxy (10). The calls are routed through the stub and skeleton to the server proxy (the RMI server) and from there to the application server (dotted line).

Of all the generated classes, only two are really interesting for a developer:

- ❑ The server proxy (4) is the actual RMI server. It includes the RMI code to set up a security manager, register itself with the RMI registry, and start the application server.
- ❑ The client proxy bean (5) is the class that stands in for the server bean. It is used to develop an applet or application by coding methods or by using the Visual Composition Editor. The client proxy bean includes the RMI code to set up a security manager and look up the server.

Created Classes and Interfaces

The RMI Access Builder takes as input an existing application server class. You specify the name of the generated client proxy bean; all other class names are derived from that.

The classes and interfaces generated by the RMI Access Builder appear in the following list (we have specified MyProxy as the name of the client proxy bean):

MyProxy: The name of the client proxy bean.

MyProxyBeanInfo: This BeanInfo class contains information about the client proxy bean interface. Note that the client proxy is a real Java bean that can be used in the Visual Composition Editor.

MyProxyS: The server proxy, that is, the actual RMI server class

MyProxyIf: This class defines the public interface of the server object. It encapsulates the public methods of the server object. The class is required by the RMI services; it is one of the supporting classes needed by the server and client proxies.

MyProxy_ListenerInterfaceRmiIf: For each event class that the server object supports, a corresponding interface class is generated, for example, MyProxyChangedListenerRmiIf. This interface supports the transmission of events from the server proxy to the client proxy. The number of interfaces generated depends on the number of event listeners involved with the client-side and server-side server proxies. The application does not have to use this interface directly.

MyProxyS_Stub: This is the RMI stub class generated by the RMI compiler from the server proxy class. The stub class works with the RMI services to communicate client requests to the server.

MyProxyS_Skel: This is the RMI skeleton class generated by the RMI compiler from the server proxy class.

MyProxy_Stub and **MyProxy_Skel:** These are RMI stub and skeleton classes generated by the RMI compiler from the client proxy class. They are needed to enable the server object to call back to the client. They are generated only if the server object produces events and has event listener registration methods. You do not have to use these classes directly, but they must be included when the application is deployed.

RMI Execution Environment with VisualAge for Java

Let us now draw the big picture to figure out how the server object, the generated beans, and the components provided by VisualAge for Java work together. First we take a look at the server side (Figure 108).

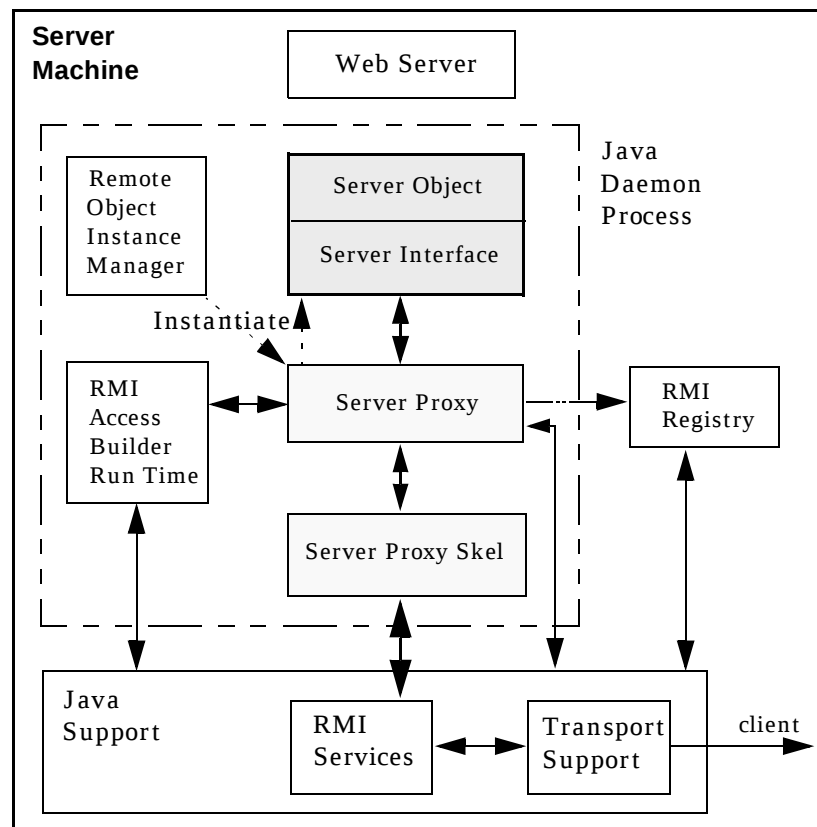


Figure 108. RMI Server Execution Environment

VisualAge for Java provides a Java Daemon process that starts the Remote Object Instance Manager.

The Remote Object Instance Manager starts the server proxy, which is responsible for instantiating the application server object and for registering itself with the RMI registry.

Calls from the client are delivered by TCP/IP to Java's Transport Support, passed through RMI Services to the server proxy skeleton object and to the server proxy. From there the original method in the server object is called, and the return value is passed back the same way.

Figure 109 shows the client machine.

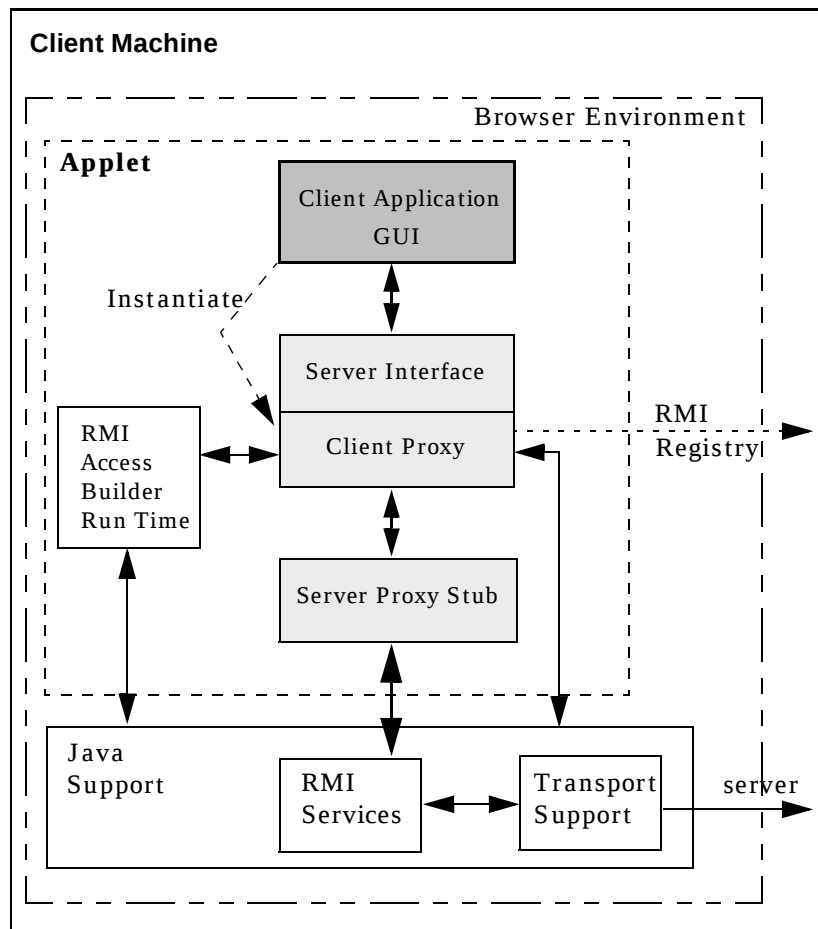


Figure 109. RMI Client Execution Environment

On the client side we have the applet that was downloaded from the Web server. According to the current Java security restrictions, the RMI server objects must run on the same machine from which the applet is downloaded.

Somewhere in the applet's code the client proxy is instantiated. The client proxy is responsible for locating the server proxy, using the RMI registry on the server machine. The client proxy forwards local method invocations to the server through the stub class, RMI services, and transport support.

Using the RMI Access Builder

Before we move on to our masterpiece, the ATM application, we create a simple applet to guide you through the steps from an ordinary application to a distributed application.

Create the Server

We use a much simplified account class as the bean to go distributed. The account has nothing but a balance property, a deposit method and a withdraw method. We create a new package called *RmiPrimer*. In this package we create an *RmiAccount* bean:

- ❑ We create a new bean in *RmiPrimer* called *RmiAccount*.
- ❑ We go to the *BeanInfo* page and add
 - A *balance* property of type *double*
 - A *void deposit(double amount)* method
 - A *void withdraw(double amount)* method
- ❑ We go to the *methods* page and code the methods:

```
public void deposit(double amount) {  
    setBalance(getBalance() + amount);  
}  
  
public void withdraw(double amount) {  
    setBalance(getBalance() - amount);  
}
```

Create an Applet

Now we create a small account applet to work with the account bean:

- ❑ We create a new applet bean called *RmiAccountApplet*
- ❑ We add two buttons labeled *Deposit* and *Withdraw*
- ❑ We add two labels, *Balance* and *Amount*.

- ❑ We add a label to display the balance, and an entry field for the amount.
- ❑ We add an RmiAccount bean to the free-form surface and connect its features to the applet's panel.

The applet should resemble that in Figure 110 and use the beans as indicated in Figure 111.

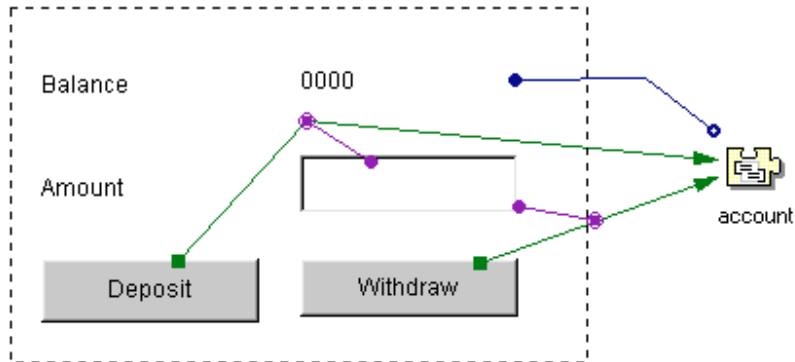


Figure 110. Account Applet before Distribution

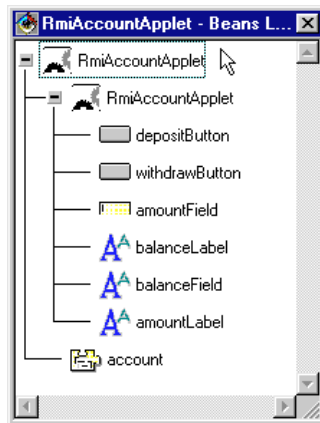


Figure 111. Beans List of Account Applet

We should be able to deposit, withdraw, and see the updated balance when running this applet.

Generate Proxy Bean

We now start the Create Proxy Bean SmartGuide by selecting *Tools -> Remote Bean Access -> Create Proxy Beans* from the RmiAccount context menu:

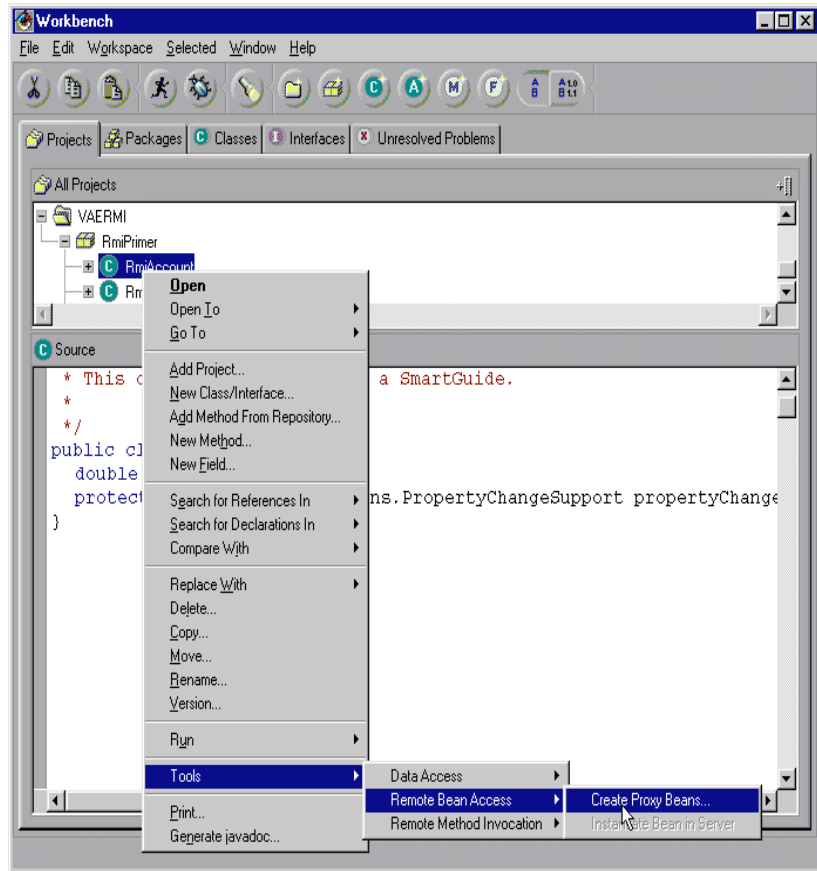


Figure 112. Start Create Proxy Bean SmartGuide

In the SmartGuide window (Figure 113) we define the name of the proxy bean as *RmiAccountProxy*. This name will be used as the prefix for the name of all generated beans.

The other entry fields define the source of the generation process and are already set to the correct values.

In this window we also find three checkboxes to indicate what we want the RMI Access Builder to generate.

- ☐ If we want to remotely access inherited methods, we select the *Include inherited methods in the proxy interface* check box. We do not need this in our example.
- ☐ If we want to generate RMI stub and skeleton classes from the generated proxy classes, we ensure that the *Create RMI stub and skeleton for generated classes* check box is selected. If this check box is not selected, we must manually run the RMI com-

piler against the server-side server proxy, and possibly against the client-side server proxy, to create the stub and skeleton classes required by the RMI run time. For this reason, we recommend that you always select this check box when generating a proxy bean.

- ❑ If we want to instantiate the generated server object, we select the *Instantiate server object* check box. This starts both the RMI registry and the Remote Object Instance Manager. We do not check this box—we consider the manual process in “Run the RMI Applet” on page 163.

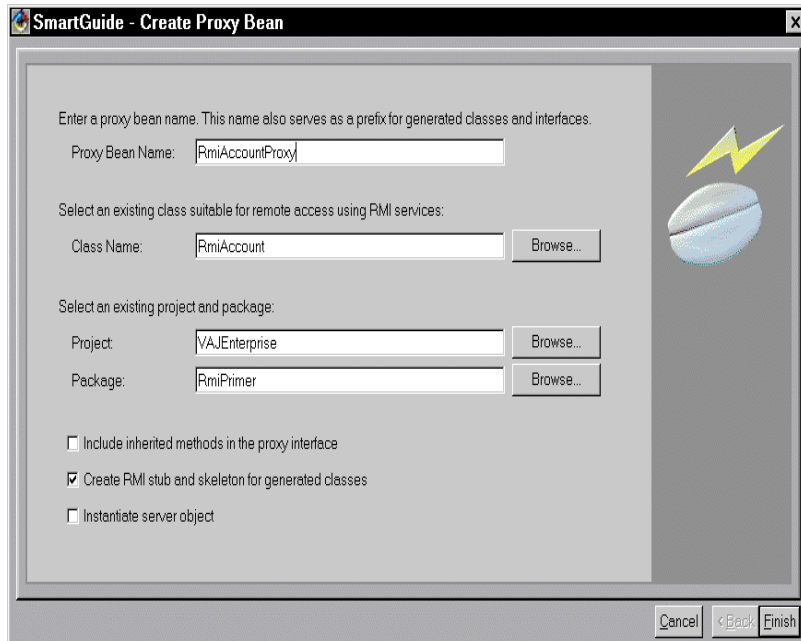


Figure 113. Create Proxy Bean SmartGuide

Now we click on **Finish** to begin generating the proxy bean and associated classes and interfaces.

Once the generation process has finished, our package has grown considerably, as you see in Figure 114:

- ❑ *RmiAccountProxy* is the client-side proxy we will use in our applet. It has an associated *BeanInfo* class that describes the bean.
- ❑ *RmiAccountProxyS* is the RMI server that implements the *RmiAccountProxyIf* interface.

For a description of the generated beans, refer to “Created Classes and Interfaces” on page 154.

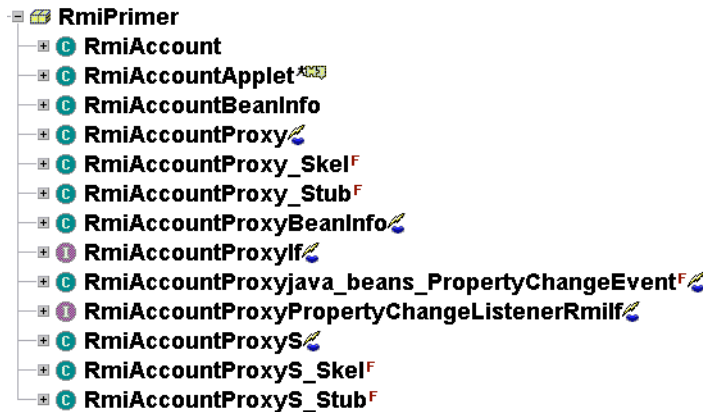


Figure 114. Generated RMI Beans

Let us investigate the `RmiAccountProxyIf`, that is, the public interface of the server. It includes these methods:

```
getBalance()
setBalance(double)
deposit(double)
withdraw(double)
addPropertyChangeListener(RmiAccountProxyPropertyChangeListenerRmiIf)
removePropertyChangeListener(RmiAccountProxyPropertyChangeListenerRmiIf)
```

Changes in the balance property of the server are propagated to each client proxy through the `propertyChange` event (actually the subclass `RmiAccountProxyjava_beans_PropertyChangeEvent`). This is why there is also a set of stub and skeleton classes generated from the client proxy.

Now we are ready to use the beans in our applet.

Connect the Client with the Server

In the `RmiAccountApplet` we can now use the new `RmiAccountProxy` bean exactly the same way we used the original `RmiAccount`. So we just add the new bean and redirect all connections to and from the `RmiAccount` to and from the `RmiAccountProxy` instead. Then we remove the old account bean (Figure 115).

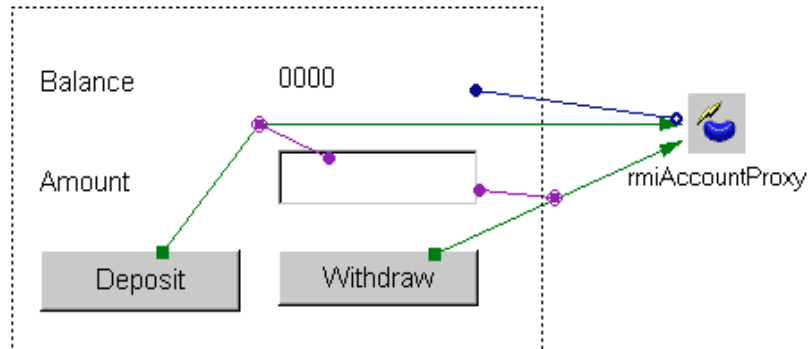


Figure 115. AccountApplet after Distribution

The changed applet looks identical to the applet we started with (Figure 110 on page 158).

Finally we have to set the Properties of the proxy bean to enable the connection to the server (Figure 116).

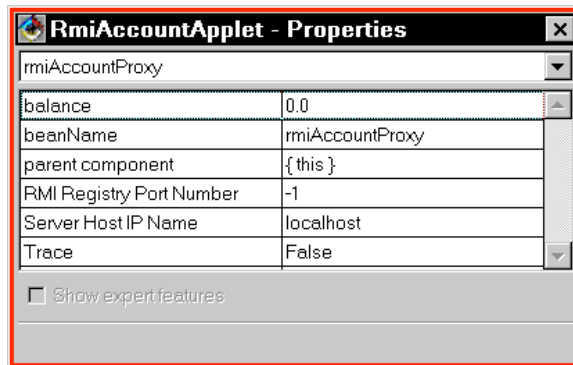


Figure 116. Properties of the RmiAccountBean

We have to set the *Server Host IP Name* and the *RMI Registry Port Number*. We use *localhost* as the host name because we run the client and server on one machine, and -1 as the port number (1099 is the default port number if -1 is specified).

Note that we also set the *parent component* property here with the value {this}. This setting is necessary because the proxy bean will be loaded with the applet and will set up the security manager. Expect to see exceptions if the parent component is not set to the applet.

The *Trace* value can be set to True to get a console trace of all RMI interactions between the client and the server.

Run the RMI Applet

Before we can actually start the applet, the server bean has to be instantiated and registered with the running RMI registry.

To start the RMI registry, select *Options* from the *Workspace* menu and go to the RMI page (Figure 117).

If we select the *Start RMI registry on VisualAge startup* check box, the RMI registry will start every time VisualAge for Java is started from now on. If we select the *Default port number* radio button, port 1099 will be used. If we change it to another port number, we have to change the properties of the proxy bean accordingly.

For a single RMI test we can also start the registry by using the **Restart RMI Registry** push button.

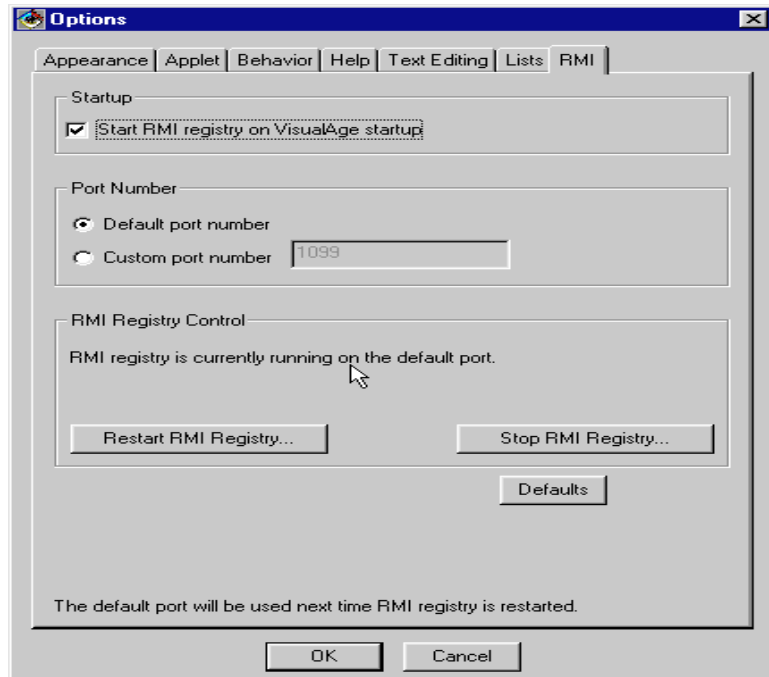


Figure 117. The RMI Options Page

Once we have started the registry, we have to create an instance of our server object.

We select the *RmiAccountBeanS* class—the generated RMI server class, not the original *RmiAccount*—and on the context menu we select *Tools -> Remote Bean Access -> Instantiate Bean in Server*.

The first time we invoke this function the Remote Object Instance Manager is started first, which takes quite some time (Figure 118).

In the lower part of its window a message eventually appears indicating that the bean has been instantiated. Only after that, can the client applet be run.

In the upper part of the window we find statistical information about the number of server objects and RMI calls. If these statistics do not show the numbers we expect, we use the *refresh* option of the *View* menu as the statistics are not refreshed automatically.

We also suggest selecting *Show Call Trace* in the *View* menu to get a trace of all RMI interactions.

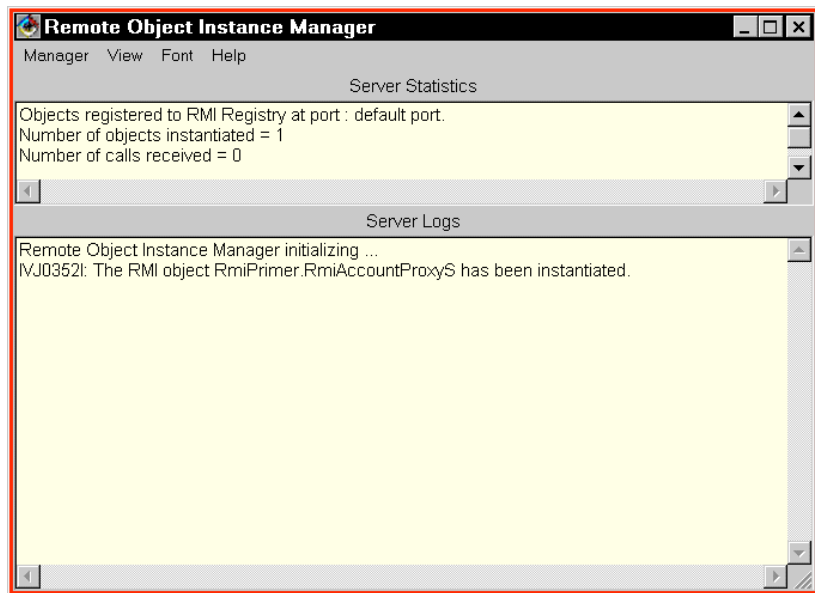


Figure 118. The Remote Object Instance Manager

Now we start the *RmiAccountApplet*—it should work as it did before the distribution.

If we start the applet twice, we can have two instances of the applet running and see from the balance displayed that both applets access the same server bean. A deposit from one applet shows up immediately in both applet windows. The property-Change event propagates any changes in the server bean's balance to all the clients (applets) that are connected to the server.

Stop the Server

When the applet has finished, we stop the server in the Remote Object Instance Manager by selecting *Remove Server Object* in the *Manager* pull-down.

RMI Problems and Hints

If your applet does not run as smoothly as described above (ours did not), here are some hints:

- ❑ The first time you access the server it may take quite a long time, even more than 20 seconds. This is normal for the first RMI call on a local machine; subsequent calls work much faster.
- ❑ You can speed up RMI if you set up TCP/IP such that host names are resolved locally instead of looking them up on a name server. If you set up your TCP/IP to run in loopback mode, do not define domain name server (DNS) entries, or create a *HOSTS* file that maps host names to TCP/IP addresses.
- ❑ To ensure that all raised exceptions are displayed in the console window, remove the `//` in the comment lines in the `handleException` method of the applet generated by VisualAge for Java:

```
private void handleException(Throwable exception) {
    /* Uncomment the following lines to print uncaught exceptions */
    // System.out.println("----- UNCAUGHT EXCEPTION -----");
    // exception.printStackTrace(System.out);
}
```

- ❑ If your applet does not work as expected, turn on the trace property of the client proxy bean. The console window will show a trace of the RMI calls:

```
IVJ0205I: RmiPrimer.RmiAccountProxy: Debug trace is set on.
IVJ0204I: RmiPrimer.RmiAccountProxy: Looking up an RMI object
RmiPrimer.RmiAccountProxy.
IVJ0206I: RmiPrimer.RmiAccountProxy: Invoking remote method:
public synchronized void addPropertyChangeListener(...).
IVJ0206I: RmiPrimer.RmiAccountProxy: Invoking remote method:
public double RmiPrimer.RmiAccount.getBalance().
IVJ0206I: RmiPrimer.RmiAccountProxy: Invoking remote method:
public void RmiPrimer.RmiAccount.deposit(double).
IVJ0206I: RmiPrimer.RmiAccountProxy: Invoking remote method:
public double RmiPrimer.RmiAccount.getBalance().
IVJ0206I: RmiPrimer.RmiAccountProxy: Invoking remote method:
public void RmiPrimer.RmiAccount.withdraw(double).
IVJ0206I: RmiPrimer.RmiAccountProxy: Invoking remote method:
public double RmiPrimer.RmiAccount.getBalance().
```

- ❑ To debug server activity, start the server trace as well (*View* menu). Sample output in the Remote Object Instance Manager looks like this:

```
IVJ0255I: Invoking RMI object -> RmiPrimer.RmiAccountProxy
IVJ0256I: RMI object (RmiPrimer.RmiAccountProxy) completed successfully
```

- ❑ The *Security Manager already set* error message indicates that the parent component of the client proxy is not properly set to be the applet. If you decide to set up the property through a visual connection, and the message appears anyway, you have a problem with the sequence in which the beans and their connections are initialized. Because this sequence is determined by methods created by VisualAge, you have to overrule the sequence by opening the property window of the client proxy and setting the value *this* directly.
- ❑ If you get the *Error creating connection to <host:port>* error message on the first invocation, or an exception indicating that a remote object cannot be found, you may not have set the RMI registry port number property to the port, on which the RMI registry is actually running on.
- ❑ If you get the *AppletSecurityException: checkconnect.networkhost1* error message when you are running the applet in the VisualAge for Java applet viewer, the reason might be in the settings of the applet viewer. Select *Properties* in the *Applet* menu and set *Network access* to *Applet Host* or *Unrestricted*.

Running an RMI Application outside VisualAge for Java

A real RMI application runs outside the development environment. Let us see how we export the applet and the server from VisualAge for Java to run stand-alone.

Export of Application Code

Select the RmiPrimer package and select *Export* in the *File* pull-down. Export the class files into a directory that is in CLASSPATH and create the package subdirectory.

In our case the d:\java directory is in CLASSPATH, and export creates the subdirectory RmiPrimer with all the class files.

Start the Registry and the Server

In a DOS (or OS/2) window we start the RMI registry and the Java daemon provided by VisualAge for Java:

```
start rmiregistry                                (start rmiregst in OS/2)
java COM.ibm.ivj.eab.rmi.javad.javad
```

The Java daemon starts the Remote Object Instance Manager. Select *Instantiate Server Object* and *By Name* in the *Manager* pull-down. Enter the name of the server as `RmiPrimer.RmiAccountProxyS` to start the server.

Run the Applet

To start the applet we need an HTML file (`RmiAccount.htm`):

```
<APPLET code=RmiPrimer.RmiAccountApplet.class width=300 height=300>
</APPLET>
```

Start the applet, using the JDK applet viewer:

```
appletviewer RmiAccount.htm
```

If the applet does not connect properly to the server, review the settings (select *Properties* in the *Applet* menu) and make sure that network access is set to *Applet Host* or *Unrestricted*.

Note that we cannot run the applet from another machine because we set the server host name to `localhost`.

Stop the applet and stop the server from the Remote Object Instance Manager window.

Before You Use RMI to Build a Distributed Application

To complete the description of RMI with VisualAge for Java, we discuss the design considerations and limitations you should be aware of if you are going to use RMI to build an application. Most of the issues originate from Java's RMI Services and not from the RMI Access Builder of VisualAge for Java.

Design Considerations

When designing RMI applications you need to consider the design of both local and remote classes and understand how parameters are passed from client to server:

- ❑ In general, server objects may reside as instances on different server machines. However, because of security constraints imposed by the Java RMI protocol, an applet can only access server objects running on the host from which the applet code is downloaded.
- ❑ Local objects are passed by copy to the server object. In a non-distribution application, objects are passed by reference in method invocations. The RMI run time only supports the passing of local objects by copy. When returning an object, the RMI

run time creates a copy of the object on the client machine. Therefore those client programs that depend on objects being passed by reference will have to be modified. More specifically: Changes made by the server object on a parameter will not be visible to the client unless the server object method returns the modified object to the client. Client programs should not use the “==” operator to check equality between a local object and an object returned from the server object. Instead, programs should use the equals() method. Note that RMI remote objects are passed by reference. An RMI remote object is an object that implements the remote interface and had RMI stubs generated for it. Client-side server proxies are also passed by reference.

- ❑ Objects passed as parameters to the server must be serializable. The RMI run time uses serialization to send parameters from the client program to the server. For this reason, any objects passed as parameters in a method invocation must implement the Serializable interface.
- ❑ System exceptions will be converted to `IVJRException`. Exceptions other than those thrown by the user will be converted to `IVJRException`. Any `RemoteException` will also be converted to `IVJRException`.
- ❑ Static methods defined in the server bean are encapsulated as nonstatic methods. Because the server bean must be instantiated before any client call is made, static methods on the server bean are provided for client invocation as nonstatic methods. You must make provisions to instantiate the client proxy before invoking methods in the server bean that were originally static.
- ❑ The library required by the Java native call must be accessible to the Remote Object Instance Manager process. The server bean runs in the Remote Object Instance Manager process space and any DLL required because of the use of native calls in the server bean must be accessible to the Remote Object Instance Manager process. You must set the PATH environment variable accordingly.
- ❑ The RMI registry must be started again when there is a change in the public interface of the server object. A change in the server bean's public interface means that the proxy bean will be regenerated. As a result of regeneration, new stub and skeleton classes are created, and the RMI registry must be restarted to pick up the new definitions. For additional information about the RMI registry, see the Sun Javasoft Website.
- ❑ The Remote Object Instance Manager may need to be restarted when there is a change in the implementation of the server bean. When running outside the Workbench, changes in the server bean's class object may not be picked up by the

Remote Object Instance Manager. This may be true even if you use the Remote Object Instance Manager to remove and then reinstantiate the server object. It appears that the class loader will not reload a class until the process is restarted. To overcome this problem, you can end the Remote Object Instance Manager and then restart it with the new implementation of the server bean.

Limitations

Here is a list of limitations you need to be aware of when implementing RMI applications:

- ❑ Server objects must be instantiated by using a default constructor. If properties must be set on the server object after instantiation, public methods must be defined in the server object to set these properties. Alternatively, the server object can be serialized to a file. The Remote Object Instance Manager will use this file to instantiate the server bean.
- ❑ Server objects must be instantiated before the client is run.
- ❑ Potential deadlock exists when a GUI client creates a modal dialog in an event action routine for an event that originates from the server object.
- ❑ Event listener action routine method signatures must be unique across all event listener interfaces referred to by the server bean. If the server bean generates several events, the method signatures of the event listener interfaces must not be repeated.
- ❑ Event listener action routines with more than one argument are not supported in the BeanInfo classes.

7

ATM Application with RMI

In Chapter 6, “Remote Method Invocation and RMI Access Builder” we explained the power of the RMI Access Builder, and constructed a sample distributed application that used the beans generated by the tool.

In this chapter we draw on that experience to use RMI to distribute the ATM application. We cover some design issues and explain some more features of the RMI Access Builder.

Design for Distribution

When we developed the first ATM application, using Data Access Builder with JDBC, we took a prototyping approach, without caring too much about design issues. When we think of creating distributed applications, we have to think about how to distribute the application, and we have to design for distribution from the beginning.

Most likely the applet will run in a Web browser on a client, access to the database will be on a server, and the actual database might be in yet another computer, such as an enterprise server. Therefore, the first step is to review the object model, restructure and expand it, and make some design decisions.

Application Layers

One prerequisite for distributed applications—and for maintainable applications in general—is a layered architecture that assigns responsibility for certain services to an appropriate application layer. These responsibilities are similar to an object model, where we assign responsibilities to each object.

First we want to separate the GUI part of the application from the core business objects. All business logic and application knowledge should be modeled in the business objects that we identified at an early stage of development (Figure 77 on page 100).

The GUI objects, however, are responsible for handling all user interactions and for presenting business objects in a nice format to users. Whenever information from the business objects is required, or an action is triggered by a user, the respective service from the business objects is called.

This separation is quite obvious in the ATM application. If the ATM application evolves over time to a real application, it is unlikely that there will be a Web browser in the bank's teller machine. Therefore, it is necessary to replace the current user interface with another one but still keep all of the business logic that is modeled in the business objects.

We also want to separate the data access from the rest of the application. Neither the GUI nor the business objects should be aware of the details about where the persistent data is stored. This separation makes it possible to have the core of the application unchanged, even if we move from one database system to another, distribute the application by using RMI, or use other services such as a transaction system to access the enterprise data.

Application Layer Architecture

We end up with three layers for the ATM application as shown in Figure 119.

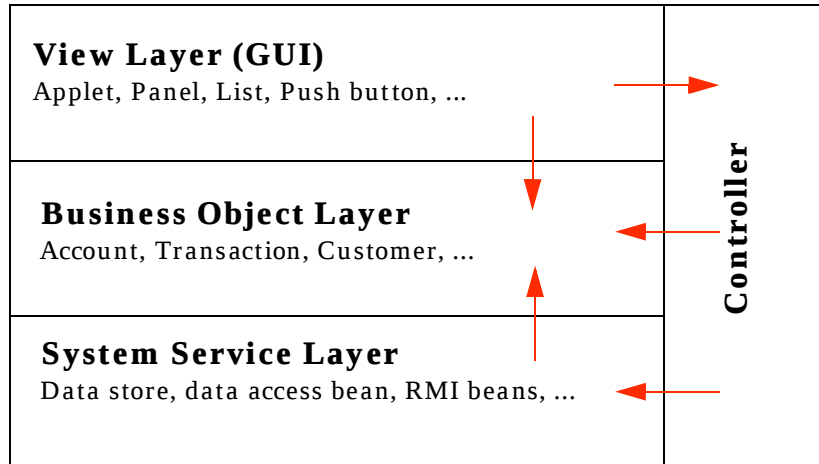


Figure 119. Layers of the ATM Application

Figure 119 also shows a cross-layer called the *controller*. The controller is necessary to connect the layers. Basically there are three ways of connecting the layers. The important principle is that there be a crisp interface among the layers.

Here are the three basic ways of connecting the layers:

- ❑ Each object has all the knowledge required to access other objects across the borders. This approach looks very appealing at first sight but can become quite cumbersome because changes at one point might affect multiple classes. Therefore, this approach is only feasible for small applications.
- ❑ A *framework* provides the necessary interfaces and underlying services. The application objects just connect to the framework, by either inheritance or delegation. This is a great approach if such a framework is available. However, the creation of a framework is difficult and requires another approach and different skills from those required for an application development project.
- ❑ The interfaces are modeled in objects that have some knowledge of both sides. Such objects are often called *mediators*. This approach has the advantage that we have only one place to update, if the interface of a layer or subsystem changes. The downside, of course, is a somewhat longer path for the messages.

We decided to use mediator objects in the ATM application; these objects are part of the controller layer. We explain their responsibilities when we describe the beans in more detail.

The most encapsulated layer is the business object layer. Business objects are not aware of the existence of the view layer, and they have limited knowledge of the data access classes. The view layer classes use the business objects but have no connection to the system service layer. The system service layer knows about the business objects, but not about the view layer.

We now give you a description of the beans in these layers. Because you are already familiar with the Java language and the VisualAge for Java development environment, we just list the features of the beans in tables, as in Table 11. We do not list the generated beans or features, and we also omit the getter and setter methods. Along with the table we give some general information on the layers and beans, where we think it is worthwhile.

Table 11. Bean Feature Table

Property	Type		Remarks
Property name	Type/class referenced		Any helpful information
Method	Return type	Parameters	Remarks
Signature	Type, class, void	Type and identifier	Any helpful information
Event	Event Interface		Remarks
Event name	Method to implement by listener		Method where and condition when event is thrown

Business Object Layer

The beans in the business object layer are similar to those in the preliminary object model (Figure 77 on page 100). However, we made some minor changes, as shown in Figure 120.

If you compare the two models, you see two main differences:

- ❑ To concur with the requirements we had to introduce a relationship between Card and Customer. This change reflects the fact that the cardholder does not have to be identical to the owner of the related account. (This change had to be reflected in the database as well.)
- ❑ We removed all information about foreign keys as they are redundant because they are already modeled through relations.

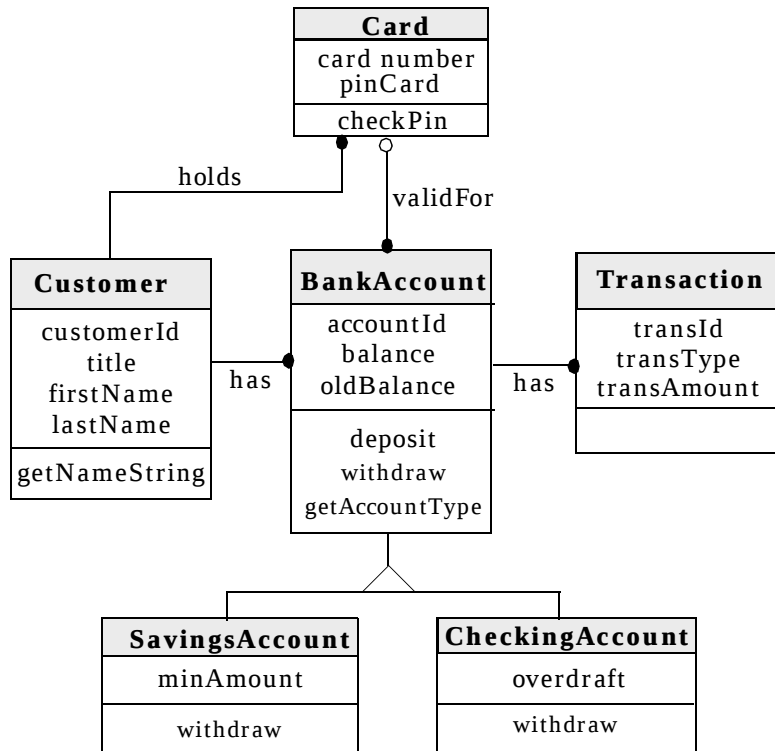


Figure 120. Object Model of the ATM Business Objects

Below we discuss the design and implementation details of the classes in the object model of the ATM business objects. We use a package named RmiModel.

BankAccount

BankAccount (Table 12) is an abstract class. The concrete classes are CheckingAccount and SavingAccount.

Table 12. BankAccount Features

Property	Type	Remarks
accountId	String	Key in the database
accountType	String	Abstract, read only
balance	BigDecimal	
oldBalance	BigDecimal	
customer	BankCustomer	Customer (account owner)
transactions	Vector	Vector of transactions

Method	Return Type	Parameters	Remarks
deposit	void	String amount	Calls deposit (BigDecimal)
deposit	void	BigDecimal amount	Updates balance and creates new transaction (private)
withdraw	boolean	String amount	Calls withdraw (BigDecimal)
withdraw	boolean	BigDecimal amount	Abstract; implemented in subclasses

Event	Event Interface	Remarks
limitExceeded	handleLimitExceeded	Thrown by withdraw if there are not enough funds

CheckingAccount

CheckingAccount (Table 13) is a subclass of BankAccount and therefore inherits all of its properties. In addition it has an overdraft property, which is the maximum amount the account is allowed to be overdrawn. If the customer attempts to withdraw an amount that exceeds this limit, a LimitExceededEvent is fired.

Table 13. CheckingAccount Features

Property	Type	Remarks
overdraft	BigDecimal	

Method	Return Type	Parameters	Remarks
getAccountType	String		Returns "Checking Account"
withdraw	boolean (1 = OK)	BigDecimal amount	Can fire limitExceededEvent

SavingsAccount

SavingsAccount (Table 14) is a subclass of BankAccount. It has an additional minAmount property, which is the minimum amount of the account's balance. If the customer attempts to withdraw an amount that would leave a balance below the limit, a LimitExceededEvent is fired.

Table 14. SavingAccount Features

Property	Type	Remarks	
minAmount	BigDecimal		
Method	Return Type	Parameters	Remarks
getAccountType	String		Returns “Savings Account”
withdraw	boolean (1 = OK)	BigDecimal amount	Can fire LimitExceededEvent

Card

The Card class can validate a PIN and knows the holder (customer) and the accounts associated with the card.

Table 15. Card Features

Property	Type	Remarks	
accounts	Vector	Vector of BankAccount	
cardNumber	String	Database key	
customer	BankCustomer	Customer (card holder)	
pinCard	String	PIN code	
Method	Return Type	Parameters	Remarks
checkPin	void	String pinEntered	Fires pinCheckedOk or pinCheckedNotOk event
getAccount	BankAccount	int index	Gets the account at position index of the accounts vector
getCustomerText	String		Calls the customer's getNameString method
Event	Event Interface	Remarks	
pinCheckedOk	handlePinCheckedOK	Thrown by checkPin if correct PIN entered	
pinCheckedNotOk	handlePinCheckedNotOK	Thrown by checkPin if incorrect PIN entered	

Customer

From the ATM application's point of view, there is hardly a reason for having a Customer class. We created one anyway to complete the model (Table 16).

Table 16. Customer Features

Property	Type	Remarks	
customerId	String	Customer identification	
firstName	String		
lastName	String		
title	String		
accounts	BankAccount	Not used, get from Card	

Method	Return Type	Parameters	Remarks
getNameString	String		Returns a formatted String with title, first and last name

Transaction

In the ATM application the Transaction objects are no more than a container of transaction data. Once they are created they are just there for reporting (Table 17).

Table 17. Transaction Features

Property	Type	Remarks	
accountId	String	Account number	
transAmount	BigDecimal		
transId	Timestamp	Database key	
transType	String	Indicates whether it is a debit or credit transaction	

Method	Return Type	Parameters	Remarks
setupDaxTrans	void	Trans daxTransaction	Creates a new transaction object for JDBC

Testing the Business Objects

After the beans (classes) for the business objects are created, they are ready for testing. If we ensure now that every bean performs the task for which it is responsible, and the beans interact properly, it is easier to locate problems—if there are problems—as we add the other layers.

To test the business objects we use the Scrapbook window, the console, and inspectors. We can support the testing task somewhat if we save test scripts for the various test cases as either files or methods. This approach can be helpful when we have to do regression testing.

We also implemented a `toString` method in each bean; it displays some of the attributes and is very handy for testing with the Scrapbook.

System Service Layer

Now that we know that the business objects are working, we can move on and create the beans for database access. You will see that there will be only minor changes to the business objects once we add the other layers.

Before we start, some preliminary remarks:

- ❑ We have in our application the ideal simple situation that one class matches one table in the database. Because this is often not the case in real-life projects, we suggest that you create views in your database system that hide joins from the applications.
- ❑ We also decided to keep all of the logic in the application, even if SQL had the capability to do more, for example, update the account only if the balance has changed.

Given the above remarks, and after revisiting the requirements (see “ATM Application Requirements” on page 18), we have the following task list for the application:

1. Retrieve the data for a Card and a BankCustomer, representing the holder of that card, based on the cardid (card number) from the CARD and CUSTOMER tables.
2. Retrieve data for all accounts based on a cardid from the ACCOUNT table.
3. Update the balance property for an account in the ACCOUNT table.
4. Retrieve a row from the ACCOUNT table with given values for accid (account number) and balance.
5. Retrieve a row from the ACCOUNT table for an accid.
6. Retrieve all transactions from the TRANS table for an account.
7. Insert a row for a new transaction into the TRANS table.

Creating the Data Access Beans

First we create a new package called `RmiDax` for the data access beans, then we use Data Access Builder to map the four ATM tables.

We start Data Access Builder for the RmiDax package by selecting *Tools -> DataAccess -> Create Data Access Beans* from the pop-up menu. We select all four ATM tables in the SmartGuide and let Data Access Builder generate the schema mappings (Figure 121).

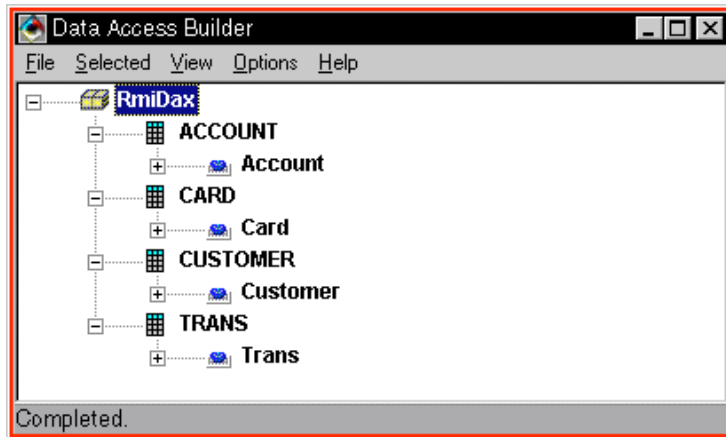


Figure 121. Data Access Builder Mappings for RMI ATM Application

Tailoring the Data Access Beans

We customize the mappings by adding user-defined SQL statements to implement the actions necessary for the ATM application.

Account

We tailor the account mapping by removing unnecessary methods and adding the extra SQL statements:

- ☐ In the Methods window (select *Methods...* from the pop-up menu) we hide the add and delete methods because they are not used in the application; the retrieve method retrieves a row from the account table (requirement 5).
- ☐ To update the balance property (requirement 3), we add a method called `updateBalance` with this SQL statement:

```
update ACCOUNT set BALANCE = ? where ACCID = ?
```

After validating the SQL statement, we change the parameter names to `balance` and `accId` and save the new method with **OK**.

- ☐ To retrieve a row with given account ID and balance (requirement 4), we add a `checkBalance` method with this SQL statement:

```
select count(*) from ACCOUNT where ACCID = ? and BALANCE = ?
```


We change the parameter names to `accId` and `balance`, and the result parameter to `count`, click on **OK**, and close the **Methods** window. The method that will be created might not be what you expect; it is a class method, and the result has to be passed as an array parameter in the call to the method. Here is the signature:

```
public static void checkBalance(String accountId,
                               BigDecimal balance, int resultC[])
    throws DAEException
```

- To retrieve all of the accounts for one ATM card (requirement 2), we select *Manager - Methods...* in the pop-up to open the **AccountManager Methods** window and add a method called `getAccountsForCard` with this SQL predicate:

```
where T1.CARDID = ?
```

We validate the SQL and change the name of the parameter to `cardId`.

Card and Customer

We only need the retrieve method (requirement 1), so we can hide the add and delete methods in the **Methods** window to avoid overhead.

Trans

To add a new transaction, we use the add method (requirement 7). To retrieve all the transaction of an account (requirement 6), we add a new manager method (select *Manager methods...* in the pop-up menu) with the following SQL predicate:

```
where T1.ACCID = ?
```

We name the method `getTransactionsForAccount` and change the parameter name to `accId`.

Driver and Database Location

To select the correct DB2 JDBC driver and to specify the database location, we open the **Properties** window of each bean (select *Properties* in the pop-up menu) and in the **Access** page set:

```
Driver: COM.ibm.db2.jdbc.net.DB2Driver
URL: jdbc:db2://localhost:8888/ATM
```

We must use the *net* driver when running under VisualAge for Java, and for testing on a single machine we set the host name to *localhost*. We always use port 8888 for the samples in this document.

Generating the Beans

Now we let the Data Access Builder do its work and select *Save and Generate All* from the *File* menu. The RmiDax package becomes populated with the data access beans.

We also get the default visual beans and the access application generated, but we will create our own GUI and have no need for them.

Initialize Business Objects from Data Access Beans

The data access beans can be viewed as object wrappers around database or result table rows. We assign the responsibility to map the data from the database row to the property fields of the business object beans to the business object. Therefore, we add a constructor to each business object that takes a Data Access Builder persistent object as input and initializes the properties of the business object.

BankAccount

```
public BankAccount (RmiDax.Account daxAccount) {
    setAccountId(daxAccount.getAccid());
    setBalance(daxAccount.getBalance());
    setOldBalance(daxAccount.getBalance());
}
```

CheckingAccount

```
public CheckingAccount (RmiDax.Account daxAccount) {
    super(daxAccount);
    setOverdraft(daxAccount.getOverdraft());
}
```

SavingsAccount

```
public SavingsAccount (RmiDax.Account daxAccount) {
    super(daxAccount);
    setMinAmount(daxAccount.getMinamt());
}
```

Customer

```
public Customer (RmiDax.Customer daxCustomer) {
    setCustomerId(daxCustomer.getCustid());
    setFirstName(daxCustomer.getFname());
    setLastName(daxCustomer.getLname());
    setTitle(daxCustomer.getTitle());
}
```

Card

```
public Card (RmiDax.Card daxCard) {
    setCardNumber(daxCard.getCardid());
    setPinCard(daxCard.getPin());
}
```

Transaction

```
public Transaction (String accountid, String transtype,
                    java.math.BigDecimal amount) {
    setAccountId(accountid);
    setTransType(transtype);
    setTransAmount(amount);
    setTransId( new java.sql.Timestamp( System.currentTimeMillis() ) );
}
```

When adding a tailored constructor, we make sure that each bean also has a default constructor without any parameter.

Creating Transaction Data Access Beans from Business Objects

Transactions are the only database objects created in our application. Therefore the transaction object also gets a `setupDaxTrans` method to initialize a persistent object from the business object.

```
public void setupDaxTrans (RmiDax.Trans daxTransaction ) {
    daxTransaction.setAccid(getAccountId());
    daxTransaction.setTransid(getTransId());
    daxTransaction.setTranstype(getTransType());
    daxTransaction.setTransamt(getTransAmount());
    return;
}
```

Connecting the Layers

The next step is to connect the layers.

Data Access Classes for Business Objects

First we create a class (in the RmiModel package) for each business object that provides a well-defined interface to the rest of the application. We call these classes CustomerDB, CardDB, AccountDB, and TransactionDB (Figure 122).

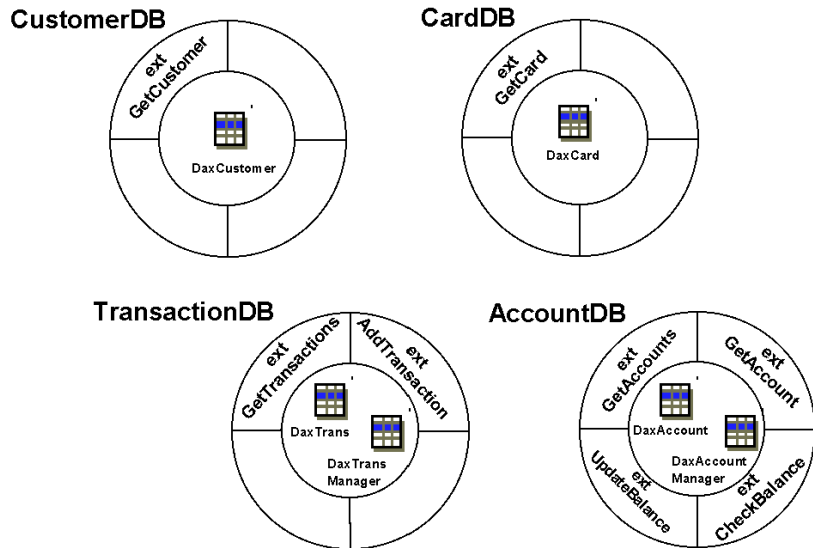


Figure 122. Data Access Classes for Business Objects

We explain these beans in the sections that follow, along with additional information that is not obvious from the picture.

For testing purposes we create these classes visually, as subclasses of `Panel`. Then we add the public methods to use the classes in a nonvisual way. All of the data access beans throw a `DAException` if something goes wrong during a database access. Therefore, when invoking a database access method, we always connect the `exceptionOccured` event to the `handleException` method of the bean, passing the event data as a parameter.

The application uses the public methods to perform the database functions. For clarity, we used the prefix `ext` for these methods, for example, `extGetCustomer`.

Customer Access Class

The CustomerDB class retrieves a customer row for a given customer ID from the database and constructs the business object (Figure 123).

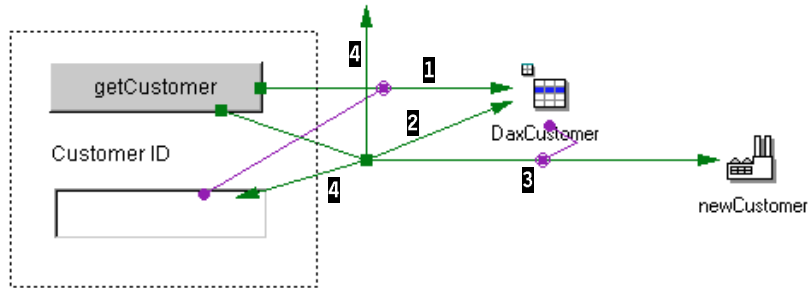


Figure 123. Visual Composition of CustomerDB Panel

Beans:

- ❑ DaxCustomer is of type RmiDax.Customer; its objectsDatastore property is promoted as datastore.
- ❑ newCustomer is a factory bean of type BankCustomer; its this property is promoted as newCustomer.
- ❑ The text property of the text field labeled Customer ID is promoted as customerId.

Connections:

- ❑ From the getCustomer button to setCustid of DaxCustomer, the text of the customer Id field is passed as a parameter (1).
- ❑ From the getCustomer button to retrieve of DaxCustomer (2).
- ❑ The normalResult is connected to the Customer(RmiDax.Customer) constructor of the customer factory, with the DaxCustomer this property as a parameter (3).
- ❑ The exceptionOccurred is connected to the handleException method and sets the customer ID to "bad" (4).

Public Method:

- ❑ The extGetCustomer method performs the action of the push button, using Java coding:

```
public RmiModel.Customer extGetCustomer(String customerId) {
    getDaxCustomer().setCustid(customerId);
    try { getDaxCustomer().retrieve(); }
    catch (Exception e) { handleException(e); }
    return new RmiModel.Customer(getDaxCustomer());
}
```

Card Access Class

The CardDB class retrieves a card row for a given card ID from the database and constructs the business object (Figure 124).

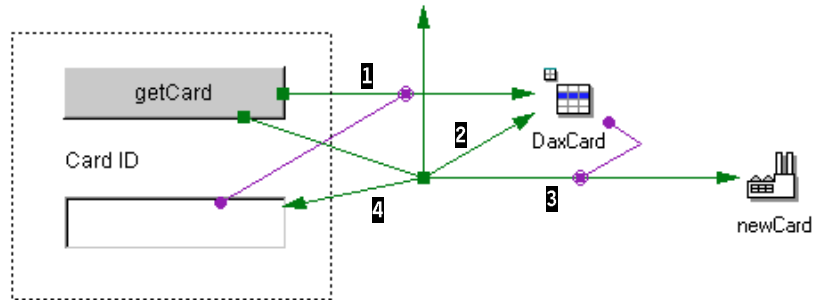


Figure 124. Visual Composition of CardDB Panel

Beans:

- ❑ DaxCard is of type RmiDax.Card; its objectsDatastore property is promoted as datastore, and custid as daxCustid.
- ❑ newCard is a factory of type Card, its this property is promoted as newCard.
- ❑ The text property of the text field labeled Card ID is promoted as CardId.

Connections:

- ❑ From the getCard button to setCardid of DaxCard, the text of the card Id field is passed as a parameter (1).
- ❑ From the getCard button to retrieve of DaxCard (2).
- ❑ The normalResult is connected to the Card(RmiDax.Card) constructor of the card factory, with the DaxCard this property as a parameter (3).
- ❑ The exceptionOccurred is connected to the handleException method and sets the card ID to “bad” (4).

Public Method:

- ❑ The extGetCard method performs the action of the push button, using Java coding:

```
public RmiModel.Card extGetCard (String cardId) {
    getDaxCard().setCardid(cardId);
    try { getDaxCard().retrieve(); }
    catch (COM.ibm.ivj.eab.data.DAException e1) { return null; }
    catch (Exception e2) { handleException(e2); }
    return new RmiModel.Card(getDaxCard());
}
```

Account Access Class

The AccountDB class retrieves all bank accounts for a given card ID or one bank account for a given account ID or updates the balance property of an account (Figure 125).

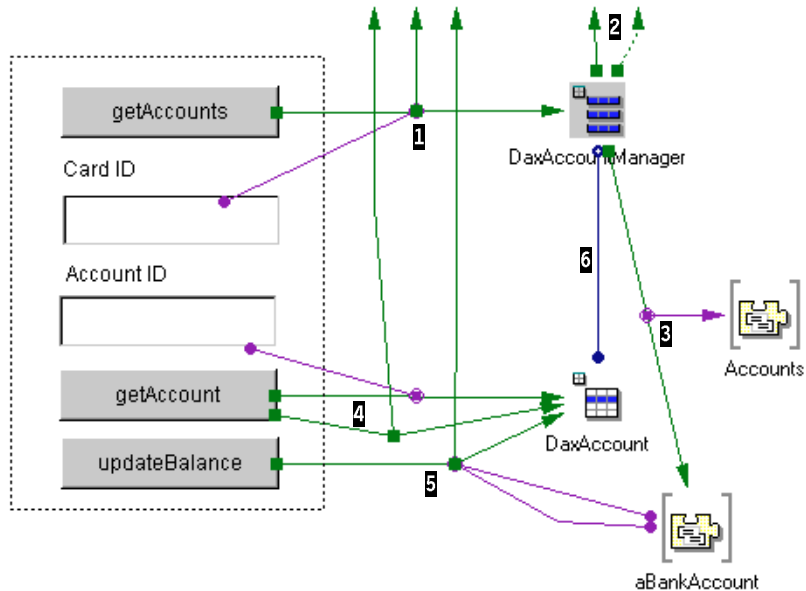


Figure 125. Visual Composition of AccountDB Panel

Beans:

- ☐ DaxAccountManager is of type RmiDax.AccountManager; its objectsDatastore property is promoted as datastore.
- ☐ DaxAccount is of type RmiDax.Account.
- ☐ Accounts is a variable of type Vector and is promoted as accounts.
- ☐ aBankAccount is a variable of type BankAccount and is promoted as aBankAccount.
- ☐ The text property of the text field labeled Card ID is promoted as accountCardId.

Connections:

- ☐ From the getAccounts button to selectGetAccountsForCard of DaxAccountManager, the text of the Card Id field is passed as a parameter (1).

- ❑ From the selectComplete event of DaxAccountManager to the extCreateAccounts method (to create the accounts vector) and to the setAccountId method with a null string (2).
- ❑ From the selectComplete event of DaxAccountManager to this of aBankAccount, getting the parameter from the elementAt(0) method of the accounts vector (3).
- ❑ From the getAccount button to setAccid of DaxAccount with the account ID as a parameter and to retrieve of DaxAccount (4).
- ❑ From the updateBalance button to updateBalance of DaxAccount, accountId and balance of aBankAccount are passed as parameters (5).
- ❑ From the objectsDatastore property of DaxAccountManager to objectsDatastore of DaxAccount (6).

Public Methods:

- ❑ The extGetAccounts method retrieves all accounts for a given card ID, calls the private extCreateAccounts method, and stores the accounts vector into the card bean that is passed as an argument:

```
public void extGetAccounts (RmiModel.Card card) {
    try {getDaxAccountManager().selectGetAccountsForCard
        (card.getCardNumber(), null);
        extCreateAccounts();
        card.setAccounts( getAccounts() ); }
    catch (Exception e) { handleException(e); }
}
```

- ❑ The extUpdateBalance method updates the account row with the new balance:

```
public void extUpdateBalance (RmiModel.BankAccount account) {
    try { getDaxAccount().updateBalance
        ( account.getBalance(), account.getAccountId() ); }
    catch (Exception e) { handleException(e); }
}
```

- ❑ The extCheckBalance method checks whether the database has the same balance stored as the business object:

```
public boolean extCheckBalance (RmiModel.BankAccount account) {
    int result[] = new int[1];
    try { getDaxAccount().checkBalance(
        account.getAccountId(), account.getBalance(), result); }
    catch (Exception e) { handleException(e); }
    return ( 1 == result[0] );
}
```


- ❑ The `extGetAccount` method retrieves one account row for a given account ID, calls the private `extCreateAccount` method, and returns the business object created:

```
public RmiModel.BankAccount extGetAccount (String accountId) {
    try {getDaxAccount().setAccid(accountId);
        getDaxAccount().retrieve(); }
    catch (Exception e) { handleException(e); }
    return extCreateAccount( getDaxAccount() );
}
```

Private Methods:

- ❑ The `extCreateAccounts` method creates the accounts vector from the `DaxAccount` items retrieved by the `DaxAccount Manager`:

```
private void extCreateAccounts ( ) {
    COM.ibm.ivj.javabeans.IVector accountRows =
        getDaxAccountManager().items();
    java.util.Vector accountVector = new java.util.Vector();
    for (int row = 0; row < accountRows.size(); row++) {
        RmiModel.BankAccount newAccount = extCreateAccount
            ( (RmiDax.Account)accountRows.elementAt(row) );
        accountVector.addElement( newAccount );
    }
    setAccounts(accountVector);
}
```

- ❑ The `extCreateAccount` method creates a checking or savings account from an account row:

```
private RmiModel.BankAccount extCreateAccount
    (RmiDax.Account accountRow) {
    RmiModel.BankAccount newAccount;
    if ( accountRow.getAcctype().trim().equalsIgnoreCase("C") )
        newAccount = new RmiModel.CheckingAccount( accountRow );
    else newAccount = new RmiModel.SavingsAccount( accountRow );
    return newAccount;
}
```

Transaction Access Class

The TransactionDB class retrieves all transactions for a given account or adds a new transaction to the database (Figure 126).

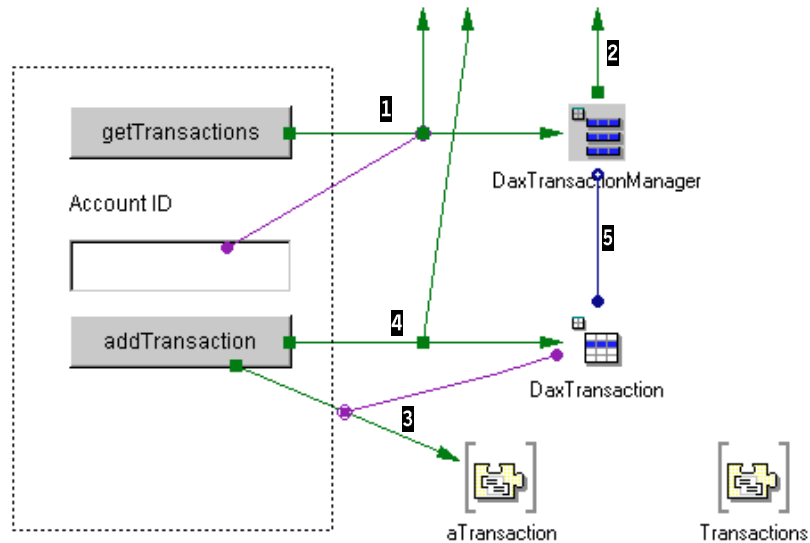


Figure 126. Visual Composition of TransactionDB Panel

Beans:

- ☐ DaxTrans is of type RmiDax.Trans.
- ☐ DaxTransManager is of type RmiDax.TransManager; its objectsDatastore property is promoted as datastore.
- ☐ Transactions is a variable of type Vector and is promoted as transactions.
- ☐ aTransaction is a variable of type Transaction.
- ☐ The text property of the text field labeled Account ID is promoted as transactionAccountId.

Connections:

- ☐ From the getTransactions button to selectGetTransactionsForAccount of DaxTransManager, the text of the account ID field is passed as a parameter (1).
- ☐ From the selectComplete event of DaxTransManager to the extCreateTransactions method (2).
- ☐ From the addTransaction button to setupDaxTrans of aTransaction, the this of DaxTrans is passed as a parameter (3).
- ☐ From the addTransaction button to add of DaxTrans (4).

- ❑ From the objectsDatastore property of DaxTransManager to objectsDatastore of DaxTrans (5).

Public Methods:

- ❑ The extGetTransactions method retrieves all transactions for an account and calls the private extCreateTransactions method to create a vector of business objects:

```
public java.util.Vector extGetTransactions
(RmiModel.BankAccount account) {
    try { getDaxTransactionManager().selectGetTransactionsForAccount
        ( account.getAccountid(), null); }
    catch (Exception e) { handleException(e); }
    extCreateTransactions();
    return getTransactions();
}
```

- ❑ The extAddTransaction method creates the DaxTransaction object from a transaction business object and adds it to the database:

```
public void extAddTransaction (RmiModel.Transaction transaction) {
    transaction.setupDaxTrans( getDaxTransaction() );
    try { getDaxTransaction().add(); }
    catch (Exception e) { handleException(e); }
}
```

Private Method:

- ❑ The extCreateTransactions method creates a vector of transactions objects from the rows retrieved by the DaxTransManager:

```
private void extCreateTransactions ( ) {
    COM.ibm.ivj.javabeans.IVector transRows =
        getDaxTransactionManager().items();
    java.util.Vector transVector = new java.util.Vector();
    for (int row = 0; row < transRows.size(); row++) {
        transVector.addElement ( new RmiModel.Transaction
            ( (RmiDax.Trans)transRows.elementAt(row) ) );
    }
    setTransactions(transVector);
}
```

ATM Service Bean

Now we can put the parts of the puzzle together and create a bean that delivers all services concerning the ATM database. We call this bean *AtmDB* (Figure 127).

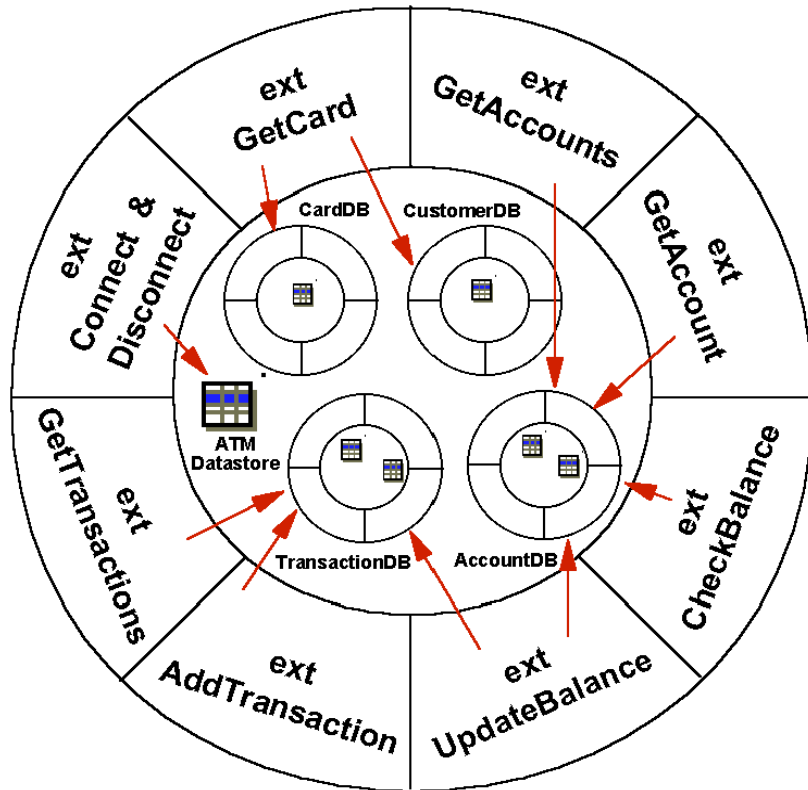


Figure 127. The *AtmDB* Bean

Because we want to use the bean for testing, we create it as an applet, and we drop the four database beans we just created onto the main panel of *AtmDB*.

All variable beans on the free-form surface (Figure 128) are created by using the *Tear-Off Property...* function against one of the data access beans.

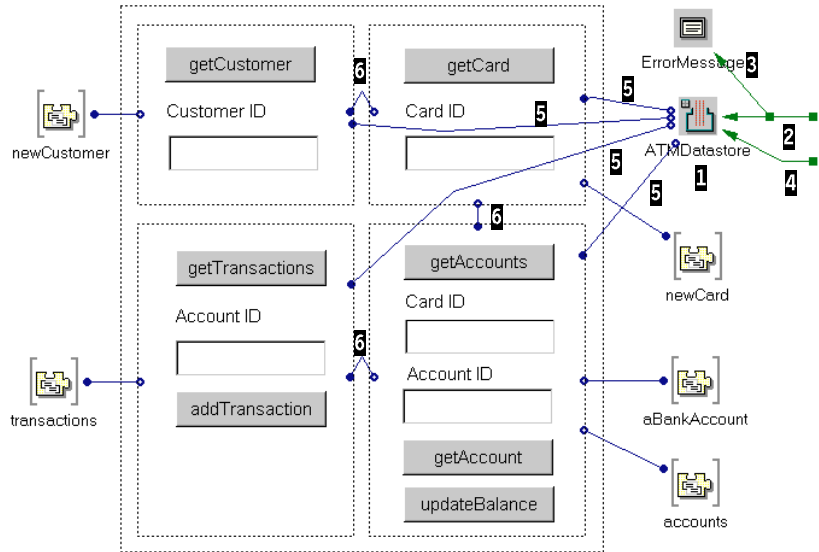


Figure 128. AtmDB Visual Composition

The important new bean is the one labeled *ATMDatastore* (1). Data Access Builder created a datastore bean for each table, for example, *CardDatastore* and *CustomerDatastore*. Because our tables all reside in the same database, we drop one of these datastore beans on the free-form surface and name it *ATMDatastore*.

The datastore bean is responsible for the connection to the database and for providing the appropriate database driver. Therefore we set the driver and URL value in the beans property, if we did not set them before generating the beans in Data Access Builder. For our environment the name of the driver is *COM.ibm.db2.jdbc.net.DB2Driver* and the URL of our database is *jdbc:db2://localhost:8888/ATM*.

To ensure that our application is connected to the database, we draw a connection from the applets init event to the connect method of the *ATMDatastore* (2). This method expects password and username as parameters. For the time being we add these as constants “userid” and “password” by using the *Set parameters* function in the Connections Properties window. We also catch the *exceptionOccurred* event and display the event in a message box (3). We also connect the applet’s *destroy* event to the *disconnect* method (4).

Finally we have to ensure that our database access beans can actually access the database. Remember that we promoted the datastore property in all of our data access beans. Now we connect the this property of the ATMDDatastore to the datastore property in each of the four subpanels (5).

The remaining connections (6) are used to test the database access; we assume you can figure out how to do that. (We connect the promoted card ID fields from the card panel to the customer and account panels, and we connect the account ID from the account panel to the transaction panel.)

The main purpose of this part, however, is to be the single entry point for all services of our application that need access to the database. The GUI part of this bean is only for some basic testing of database access.

To make the AtmDB bean a real service bean, we have several simple methods:

- ❑ The `extConnect` methods initializes the applet, for example it connects to the database:

```
public void extConnect ( ) {  
    init();  
    return;  
}
```

- ❑ The `extDisconnect` method calls the applet's destroy method.
- ❑ The `extGetCard` method creates the card and customer objects from the database, connects the customer to the card, and returns the card:

```
public RmiModel.Card extGetCard (String cardid) {  
    RmiModel.Card card = getCardDB().extGetCard(cardid);  
    if ( card != null ) {  
        String custid = getCardDB().getDaxCardCustid();  
        RmiModel.Customer customer =  
            getCustomerDB().extGetCustomer(custid);  
        card.setCustomer(customer);  
    }  
    return card;  
}
```

- ❑ The remaining methods, `extGetAccounts`, `extGetAccount`, `extUpdateBalance`, `extCheckBalance`, `extGetTransactions`, and `extAddTransaction` redirect the call to the same named method of an internal bean.

These methods can be created by promoting the matching method of the internal bean (AccountDB and TransactionDB) in the Visual Composition Editor, or they can be created manually:

```
public void extGetAccounts(Card card) {
    getAccountDB().extGetAccounts(card);
}

public BankAccount extGetAccount(String accountId) {
    return getAccountDB().extGetAccount(accountId);
}

public void extUpdateBalance(BankAccount account) {
    getAccountDB().extUpdateBalance(account);
}

public boolean extCheckBalance(BankAccount account) {
    return getAccountDB().extCheckBalance(account);
}

public java.util.Vector extGetTransactions(BankAccount account) {
    return getTransactionDB().extGetTransactions(account);
}

public void extAddTransaction(Transaction transaction) {
    getTransactionDB().extAddTransaction(transaction);
}
```

Now our database access subsystem is complete. Once it is tested, we do not have to touch it as long as the database remains the same and we do not need additional services from the database. Later, when we use RMI to distribute the database access, there will be no side effects on this subsystem.

Controller

The next step is to create *the bean*, the one that glues the layers together and will be the starting point for implementing RMI. We call this bean `ATMApplicationController`, the interface between the business model and the GUI. This bean controls what is happening inside the application, but, if we have a closer look at the internals, we see that the controllers main task is to delegate the work. Our model seems to be quite close to real life.

Controller Features

We create the controller bean as a subclass of `Object` (in the `Rmi-Model` package), with the design outlined in Table 18.

Table 18. Controller Features

Property	Type	Remarks	
atmDB	AtmDB	Database interface, read-only property	

Method	Return Type	Parameters	Remarks
connect	void	-	Initialize AtmDB and connect to the ATM database
disconnect	void	-	Destroy AtmDB and disconnect from the database
getCard	void	String cardId	Construct card object from the database, fire cardFound or cardNotFound event
getAccounts	Card	Card	Retrieve the accounts of the card, construct the accounts vector in the card, return the card
deposit	BankAccount	BankAccount, String amount, Card	Deposit amount, update balance, create new transaction, fire newTransaction event
withdraw	BankAccount	BankAccount, String amount, Card	Check balance against database, throw DBOutOfSynch event if unequal. Try to withdraw amount. If OK, update balance, create new transaction, fire newTransaction event
getTransactions	BankAccount	BankAccount, Card	Retrieve all transactions, create objects, store in account

Event	Event Interface	Remarks
cardFound	handleCardFound	Thrown by getCard if valid card number entered
cardNotFound	handleCardNotFound	Thrown by getCard if invalid card number entered
newTransaction	handleNewTransaction	Thrown by deposit (always) and withdraw (if successful)
DBOutOfSynch	handleDBOutOfSynch	Thrown by withdraw if balance in database different from account object
limitExceeded	handleLimitExceeded	Propagated from bank account to controller

To start the implementation we add the atmDB property as a read-only property (it will never change).

Most requests are routed to the AtmDB and/or the business object beans; however, the AtmApplicationController checks the result where necessary and takes appropriate action. If we ever have to

access another database, database system, or another external system such as a transaction system, the references to these would also be placed in the `AtmApplicationController`.

Controller Methods and Events

Let us review the scenario of our application and implement the methods and events step by step in the bean:

- ❑ The customer enters the card number. If the number is valid, the system prompts for the PIN; otherwise an error message appears.

This is an easy task. We ask the `AtmDB` to retrieve the card, and, depending on the result, we fire an event. In the `BeanInfo` page, we add the `cardFoundEvent` and the `cardNotFoundEvent` event features. Then we add the `getCard` method feature and write the code:

```
public Card getCard(String cardid) {
    Card newCard = getAtmDB().extGetCard(cardid);
    if (newCard != null)
    { fireHandleCardFound( new CardFoundEvent(this) ); }
    else
    { fireHandleCardNotFound( new CardNotFoundEvent(this) ); }
    return newCard;
}
```

- ❑ The PIN is validated. If it is valid, the system shows a list of the accounts the card is authorized to access.

The PIN is validated by the card (`checkPin` method). The controller gets a request to retrieve the accounts only if the PIN is valid. This request is passed to the `AtmDB` where the accounts are created and added to the vector of accounts inside the card. We add the `getAccounts` method feature and write the code:

```
public Card getAccounts(Card card) {
    getAtmDB().extGetAccounts(card);
    return card;
}
```

- ❑ The customer selects one account; the system displays the account details. All information is already in the account object. No services are needed.
- ❑ The customer performs a deposit transaction for the selected account.

Another easy task. The controller calls the account bean to update itself and to create the transaction. Account and transaction are handed over to `AtmDB` to be updated or inserted. We create a `newTransaction` event so that the GUI can react and display the transaction, and we add the `deposit` method feature and write the code:

```
public BankAccount deposit(BankAccount account, String amount,
                           Card card) {
    account.deposit(amount);
    getAtmDB().extUpdateBalance(account);
    getAtmDB().extAddTransaction
        ( (Transaction)account.getTransactions().lastElement() );
    fireHandleNewTransaction( new NewTransactionEvent(this) );
    return account;
}
```

- ❑ The customer performs a withdraw transaction for the selected account.

For a withdrawal, The system performs two levels of checks. If the account has been updated in the meantime, the request is rejected and the actual state of the account is displayed. If the overdraft or minimum balance limit of the account would be reached because of the withdrawal, the request is rejected; otherwise the balance is updated and a new transaction is added to the account history.

This implementation is a bit more complex. The manager first checks with the AtmDB if the account's balance on the database has been changed since it was read. If this is the case, a DBOutOfSynch event is fired and AtmDB is asked to update the data in the account bean. If the database is still in synch, the manager asks the account to perform the withdraw action. If the withdraw was successful, the manager hands the account and new transaction over to the AtmDB to update the database. If the withdrawal was not allowed (the account fires a limitExceeded event and returns false), we propagate the event, and the manager returns the account unchanged.

We define the DBOutOfSynch event, add the withdraw method feature, and write the code:

```
public BankAccount withdraw(BankAccount account, String amount,
                             Card card) {
    //.....
    if ( getAtmDB().extCheckBalance(account) == false ) {
        java.math.BigDecimal oldBal = account.getOldBalance();
        account = getAtmDB().extGetAccount(account.getAccountId());
        account.setOldBalance(oldBal);
        fireHandleDBOutOfSynch(new RmiModel.DBOutOfSynchEvent(this));
    }
    else
        withdrawPerform(account, amount, card);
    return account;
}
```

WithdrawPerform is a private method to execute the withdraw of funds:

```

private BankAccount withdrawPerform(BankAccount account, String amount,
                                    Card card) {
    boolean ok = account.withdraw(amount);
    if (ok) {
        getAtmDB().extUpdateBalance(account);
        getAtmDB().extAddTransaction
            ( (Transaction)account.getTransactions().lastElement() );
        fireHandleNewTransaction( new NewTransactionEvent(this) );
    }
    return account;
}

```

- The customer might want to look at the transaction history.

The system retrieves all transactions of the account and stores them in the vector property. For this we add the `getTransactions` method feature and write the code:

```

public BankAccount getTransactions(BankAccount account, Card card) {
    account.setTransactions( getAtmDB().extGetTransactions(account) );
    return account;
}

```

All the above functions require a connection to the ATM database. The controller implements a `connect` method that passes the request to the `AtmDB` bean. We add a *connect* and a *disconnect* method feature to the bean and write the code:

```

public void connect () {
    getAtmDB().extConnect();
}
public void disconnect () {
    getAtmDB().extDisconnect();
}

```

Now the `AtmApplicationController` bean is ready. You might wonder why we pass the card along to most methods, even if it is not used. The reason is a last minute request by the project sponsor to be able at a later point in time to validate the card on the server in every step.

Event Propagation

Our master plan is to move the controller to a server in the future. In “Review the Events” on page 214, we explain that only events that are fired in the server bean itself are propagated back to the client proxy bean.

The `limitExceeded` event is fired by the `BankAccount` bean but must be propagated to the client. Therefore, we have to define the `limitExceeded` event as a feature of the controller as well, and have to fire the event when it occurs in the `withdraw` method of the bank account.

Here are the steps to propagate the event:

- ❑ Define the `limitExceeded` listener feature with a `handleLimitExceeded` method.
- ❑ Change the `withdraw` method (of the controller) to listen for the event:

```
public BankAccount withdraw(BankAccount account, String amount,
                           Card card) {
    account.addLimitExceededListener(this);
    if ( ...)
        .....
    else
        .....
    account.removeLimitExceededListener(this);
    return account;
}
```

- ❑ The controller is a listener for this event. We add the `LimitExceededListener` interface to the class:

```
public class ATMAplicationController
    implements RmiModel.LimitExceededListener { ...
```

- ❑ When the `limitExceeded` event is fired in the bank account, it invokes the `handleLimitExceeded` method in the controller. We implement the `handleLimitExceeded` method to fire the event, that is, we propagate it:

```
public void handleLimitExceeded(RmiModel.LimitExceededEvent event) {
    fireHandleLimitExceeded
        ( new LimitExceededEvent(this,event.getErrorMessage()) );
}
```

Note that we have to create a new event, because we have to set the controller as the source of the event.

External Interface

Figure 129 shows the final interfaces of the `AtmApplicationController`.

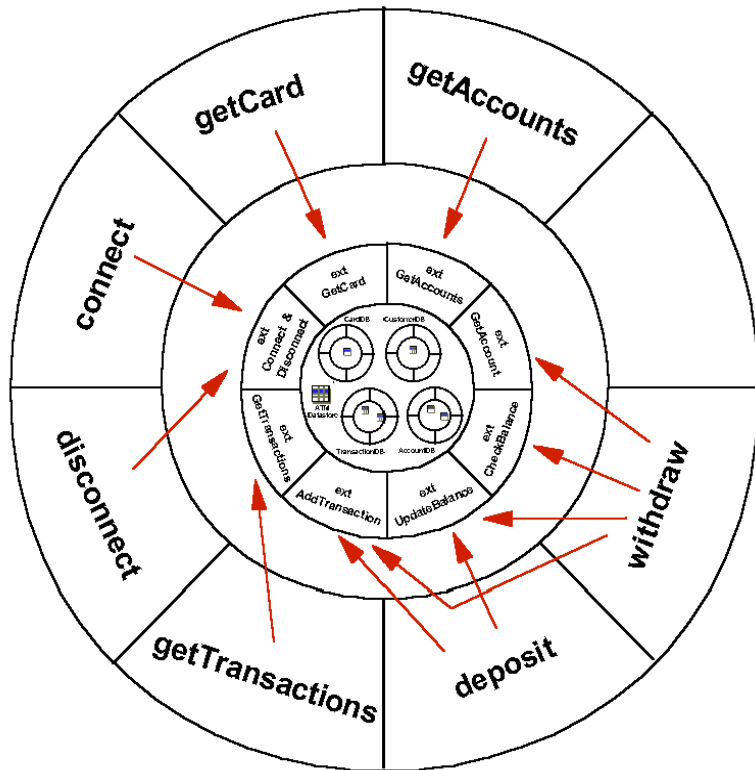


Figure 129. Application Controller Interfaces

Testing the Beans

Now the controller and business parts of the system have all the functions they need, and it is time for another test, again from the Scrapbook window.

The following script is based on the database content as listed in “Sample Data of ATM Tables” on page 30. It can be used to test the different cases, just by changing the numbers and/or commenting out lines:

```
System.out.println("----- ATM Controller Test -----");
RmiModel.ATMApplicationController control =
    new RmiModel.ATMApplicationController();

control.connect();

RmiModel.Card card = control.getCard("111111");

control.getAccounts(card);

for (int index = 0; index < card.getAccounts().size(); index++) {
    RmiModel.BankAccount account =
        (RmiModel.BankAccount)card.getAccounts().elementAt(index);
    System.out.println(account);
    control.getTransactions(account, card);
    if (account.getTransactions() != null)
        System.out.println( account.getTransactions() );
}

RmiModel.BankAccount myAccount =
    (RmiModel.BankAccount)card.getAccounts().elementAt(0);
System.out.println(" ");
System.out.println(myAccount);

control.deposit(myAccount, "101", card);
System.out.println(myAccount);

control.withdraw(myAccount, "100", card);
System.out.println(myAccount);

control.disconnect();
```

The business side of the model is done. Now let us look at the user interface.

View Layer

Now that we are sure that our core application works as requested, we are ready to connect it to the user interface. We reuse as much as possible from Chapter 5, “ATM Application with Data Access Builder and JDBC” (see “User Interface Classes” on page 115). We create the classes in a new package named `RmiGui`.

We have to fit in our new `AtmApplicationController` and change some connections accordingly. As we touch the beans, we also rearrange some connections to reach a higher degree of encapsulation.

You remember the structure of the applet as we left it (Figure 130, left side). We keep the structure of the main panel but remove all of the beans from the free-form surface and replace them with beans suited for the distributed design (Figure 130, right side).

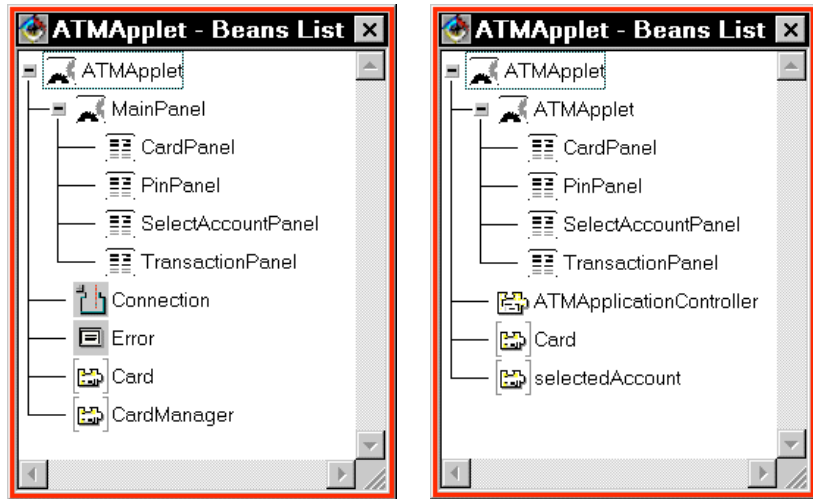


Figure 130. ATM Applet Beans List: Old and New

We discuss the beans on the main panel later. Let us first concentrate on the common changes on all of the subpanels:

- ❑ We remove all existing beans from the free-form surface.
- ❑ Each subpanel gets a variable bean called *Controller* of type *AtmApplicationController*, and its this property gets promoted as *controller*. This is the bean that provides the database services.
- ❑ Each panel gets a variable bean called *Card* of type *Card*, and its this property gets promoted as *card*. The card is used in most calls to the controller.
- ❑ Each panel gets a Variable bean called *LayoutManager* of type *CardLayout*. The layout manager is used to switch between the four panels.
- ❑ Each panel gets a variable bean called *Main* of type *Panel*, and its this property gets promoted as *main*. The main panel is used in all calls to the layout manager (as parent).

Three of the four beans (controller, card, and main) are initialized by the main panel. The layout manager is initialized on each subpanel (connection 0 in Figure 131):

- ❑ We connect the this event of the *Main* bean with the this property of the *LayoutManager* bean and pass the layout property of *Main* as parameter. When the main variable is assigned, its layout manager property sets the layout manager bean.

Now let us have a brief look at the four subpanels.

CardPanel Class

Figure 131 shows the CardPanel.

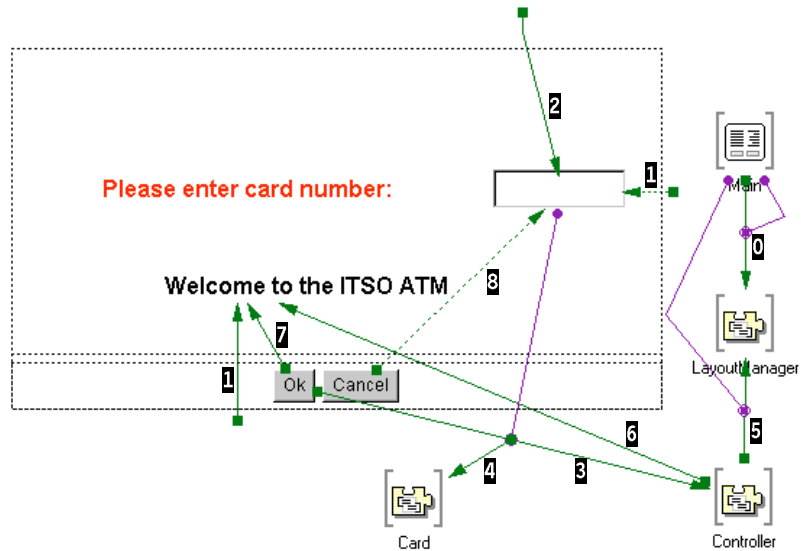


Figure 131. Visual Composition Editor: CardPanel

We initialize the welcome message and the card number entry field from the `componentShown` event of the panel (1) and request focus in the entry field (2).

The most important connection is from the **Ok** button to the `getCard` method of the controller (3), with the entry field as a parameter. We connect the `normalResult` with the `this` property of the card (4); now we have a card object for the other subpanels.

We connect the `cardFound` event of the controller with the next method of the layout manager and pass the `this` property of the main variable as a parameter (5). All calls to the layout manager have a parent parameter.

We connect the `cardNotFound` event of the controller with the message label to display an error message (6). In this case we do not switch to the next panel.

The **Ok** button also displays a “please wait” message (7), and the **Cancel** button resets the entry field (8).

PinPanel Class

Figure 132 shows the PinPanel.

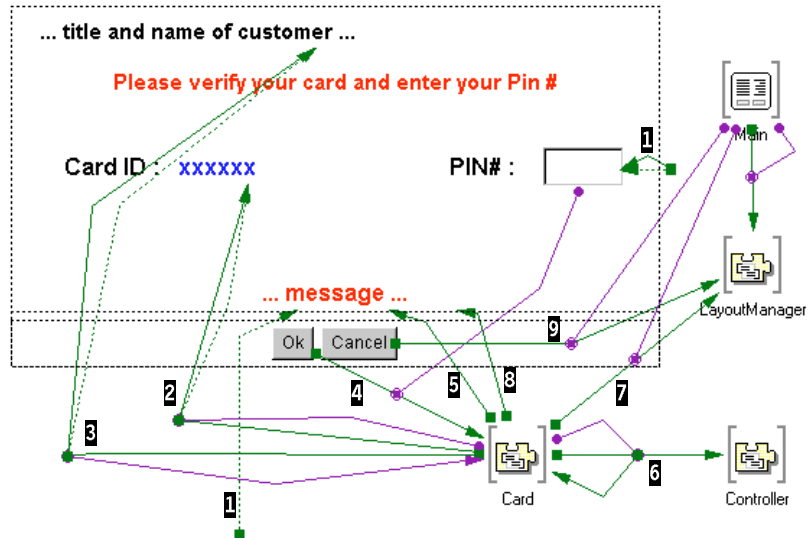


Figure 132. Visual Composition Editor: PinPanel

We initialize the message and the PIN entry field from the componentShown event of the panel and request focus in the PIN field (1).

The card bean is where all the action is. Figure 133 shows the connections of the card bean. First the card bean is used to initialize the card number and the customer text. We connect the this event of the card (when it is created) with the text of the card number (2) and use the cardNumber property as a parameter. We also connect the this event of the card with the customer welcome message (3) and use the getCustomerText method as a parameter. The dashed lines connect the exceptionOccurred result with the text labels to set them to blank.

To validate the PIN, we connect the **Ok** button with the checkPin method of the card (4) and pass the PIN entered as a parameter. This method signals either a pinCheckedNotOk or a pinCheckedOk event. We connect the pinCheckedNotOk event of the card with the message (5) and display an error (the panel will not switch).

If the PIN is valid, we have three actions. We connect the pinCheckedOk event with the getAccounts method of the controller to retrieve the accounts associated with the card (6). We pass

the card number as a parameter and connect the normalResult with the this property of the card. We also connect the pinCheckedOk event with the next method of the layout manager to switch to the select account panel (7), and we set the message to blank (8).

Finally, we connect the **Cancel** button with the previous method of the layout manager (9) to switch back to the card panel.

Card(Card) - Reorder connections			
Drag connections to reorder			
Source Bean	Source Feature	Target Bean	Target Feature
Card	pinCheckedOk	Controller	getAccounts(RmiModel.Card)
conn4: (Card.pinCheckedOk, arg1)		Card	this
Card	pinCheckedOk	LayoutManager	next(java.awt.Container)
Card	pinCheckedOk	errorMessage	text
Card	pinCheckedNotOk	errorMessage	text
Card	this	cardNumLabel	text
conn16: (Card.this --> cardNumLabel.value)		Card	cardNumber
Card	this	customerText	text

Figure 133. PinPanel: Connections from Card

SelectAccountPanel Class

Figure 134 shows the SelectAccountPanel.

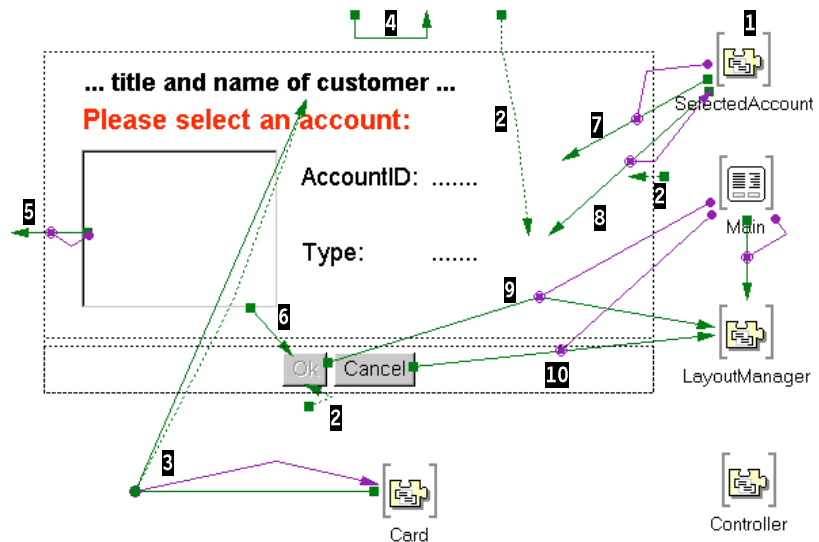


Figure 134. Visual Composition Editor: SelectAccountPanel

On this panel we need an additional variable bean on the free-form surface. The variable bean is of type `BankAccount` and named `SelectedAccount` (1). It will hold the account selected in the list box. Its this property is promoted as `selectedAccount`.

We initialize the account ID and type from the `componentShown` event of the panel and disable the **Ok** button (2). We also initialize the customer text from the `this` event of the card and get the text by calling `getCustomerText` (3), exactly as we did in the PIN panel.

The first application action is to fill the list box (named `AcctList`) with account numbers. We connect the `componentShown` event of the panel with a new method called `createAccountList` (4). The accounts are already stored in the `accounts` property of the card:

```
private void createAccountList ( ) {
    java.util.Vector accounts = getCard().getAccounts();
    String[] acctNumbers = new String[accounts.size()];
    for (int i = 0; i < accounts.size(); i++)
        acctNumbers[i] =
            ((RmiModel.BankAccount)accounts.elementAt(i)).getAccountId();
    getAcctList().setElements
        ( new COM.ibm.ivj.javabeans.IVector(acctNumbers) );
}
```

All this happens before the panel is displayed. The customer sees the list of accounts and selects one. This selection triggers the `itemStateChanged` event of the list box. We connect this event to a new method called `selectAccount`, passing the `selectedIndex` property as a parameter (5), and to the **Ok** button to enable it (6).

The `selectAccount` method sets the selected account variable bean from the account object stored in the card:

```
private void selectAccount (int index) {
    setSelectedAccount
        ((RmiModel.BankAccount)getCard().getAccounts().elementAt(index));
}
```

Two properties of the selected account are displayed on the panel. We connect the `this` event of the selected account to the account ID label passing `accountId` as a parameter (7), and to the account type label, using the `getAccountType` method as a parameter (8).

Finally, we connect the **Ok** button to the next method (9) of the layout manager to display the transaction panel, and the **Cancel** button to the first method (10) to restart at the card panel.

TransactionPanel Class

Figure 135 shows the transaction panel; it is the most complex of the application.

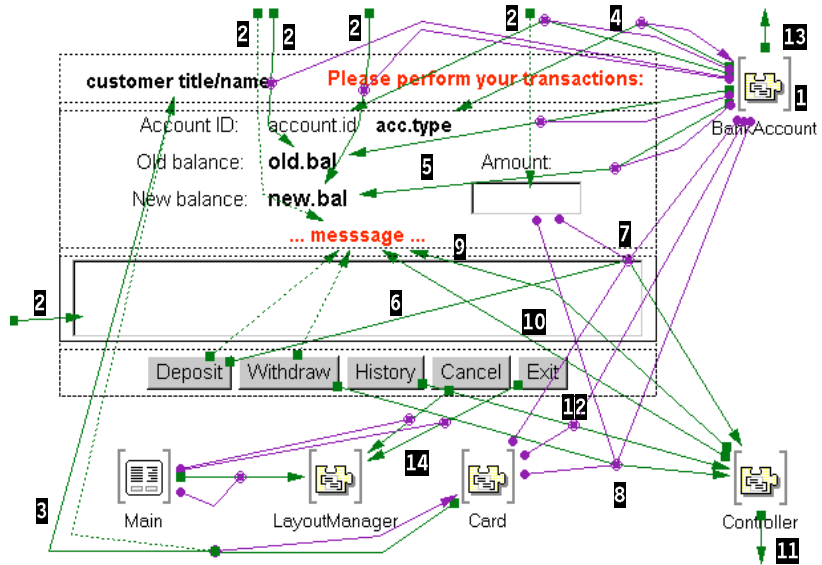


Figure 135. Visual Composition Editor: TransactionPanel

The TransactionPanel has two purposes: to display the balance information of the account, and to perform transactions against it. To display the properties is obvious for the most part; the transactions and the transaction history are a bit more demanding.

Again we need an additional variable bean of type BankAccount (1), named BankAccount, with the this property promoted as bankAccount. The variable bean holds the account selected in the previous panel.

From the componentShown event of the panel we initialize the message and amount fields, and we empty the transaction history list box using the removeAll method. We initialize the two balance fields with the information from the BankAccount (2). We also initialize the customer text from the this event of the card with the getCustomerText method (3).

We display the account ID and type on the panel from the this event of the bank account (4); exactly the same as on the previous panel. We also propagate the balance and oldBalance events of the bank account to the panel's label text (5); we pass the balance or oldBalance properties as parameters. (Note that we connect the events of the properties so that there are no initialization calls.)

Now we react to customer actions, using the push buttons. We connect the **Deposit** button with the deposit method of the controller (6), passing the bank account, amount field, and card as parameters (7). This method returns the bank account, but we disregard it in the local implementation.

The withdrawal of funds is handled similarly; however, it can fail because of insufficient funds or database synchronization.

We connect the **Withdraw** button with the withdraw method of the controller, passing the bank account, amount field, and card as parameters (8). We connect the `handleLimitExceeded` event of the controller with the message label (9) and select the *Pass event data* check box. The `limitExceeded` event contains a formatted error message that we retrieve and set as value in the *Set parameters* window:

```
" +arg1.getErrorMessage()+"
```

Note: We could also connect the `limitExceeded` event from the bank account bean, where it actually originates. We explain in “Review the Events” on page 214 that you have to capture this event at the controller bean when you move the controller to a server.

We also connect the `DBOutOfSynch` event of the controller with the message label (10) and display the text “Account data has changed - please verify - reenter transaction.”

Both deposit and withdraw methods create new transaction objects in the bank account. To display the new transactions we connect the `newTransaction` event of the controller with a new method called `buildTransactionList` (11). This method retrieves the transactions from the account and displays them in the transaction history list box (named `TransactionList`):

```
private void buildTransactionList ( ) {
    getTransactionList().removeAll();
    for (int i=getBankAccount().getTransactions().size()-1; i>=0; i--)
        getTransactionList().addItem
            (getBankAccount().getTransactions().elementAt(i).toString());
    return;
}
```

The transaction history list box displays new transactions only, unless the customer asks for the full history. We connect the **History** button with the `getTransactions` method of the controller, passing the bank account and card as parameters (12), and returning the bank account. This method re-creates the transactions property in the bank account. We connect the transactions event of the bank account with the `buildTransactionList` method of the panel to display all transactions (13).

Finally, we connect the **Cancel** button with the previous method and the **Exit** button with the first method of the layout manager (14).

Main Panel

Now we can construct the main panel (Figure 136).

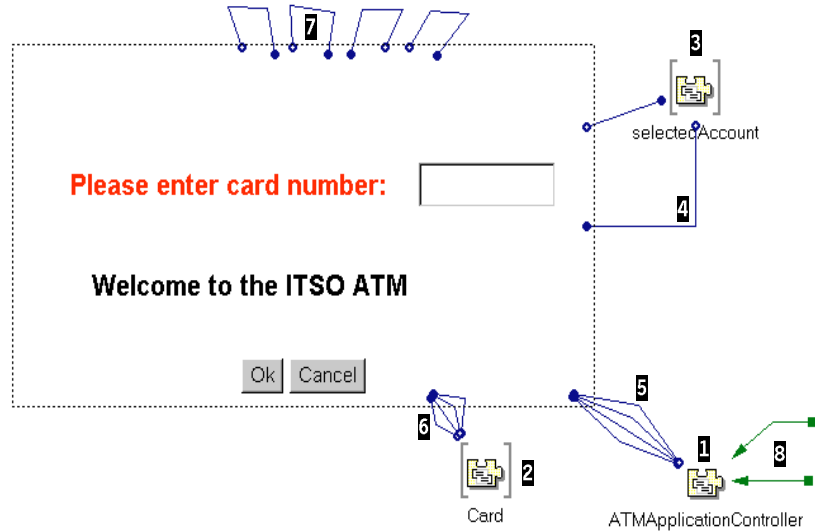


Figure 136. Visual Composition Editor: Main Panel

The main panel uses a card layout with the four subpanels in it. We need two beans, an `AtmApplicationController` (1) and a variable of type `Card` (2). (Note that the controller is not a variable.)

We open the Beans List window (Figure 130 on page 203) to switch between the subpanels; only one is displayed at a time.

The connections on the main panel pass information between the subpanels. We tear off the promoted `selectedAccount` property of the `SelectAccountPanel` (3) and connect the `this` property of `selectedAccount` with the promoted `bankAccount` property of the `TransactionPanel` (4). The direction of this connection is important because the `selectedAccount` variable gets initialized first.

Now we make sure to initialize the controller, card, and main beans on all subpanels. We connect the `this` property of the `AtmApplicationController` with the promoted controller properties of each subpanel (5). We connect the `this` property of the card variable with the card properties (6) and the `this` property of the main panel with the main properties of the subpanels (7). These last

connections are hard to draw. First display the subpanel, using the beans list, select the main panel, and then connect from it to the subpanel displayed inside.

Finally, we connect the applet's init event to the connect method and the applet's destroy event to the disconnect method of the `AtmApplicationController` (8); these connections establish and remove the database connection.

Testing the Stand-Alone Applet

Now we are ready to run the `ATMApplet` in the applet viewer and test our work.

Figure 137 shows a sample sequence of panels when the applet is run.

First tests seldom work and there are a number of potential problems:

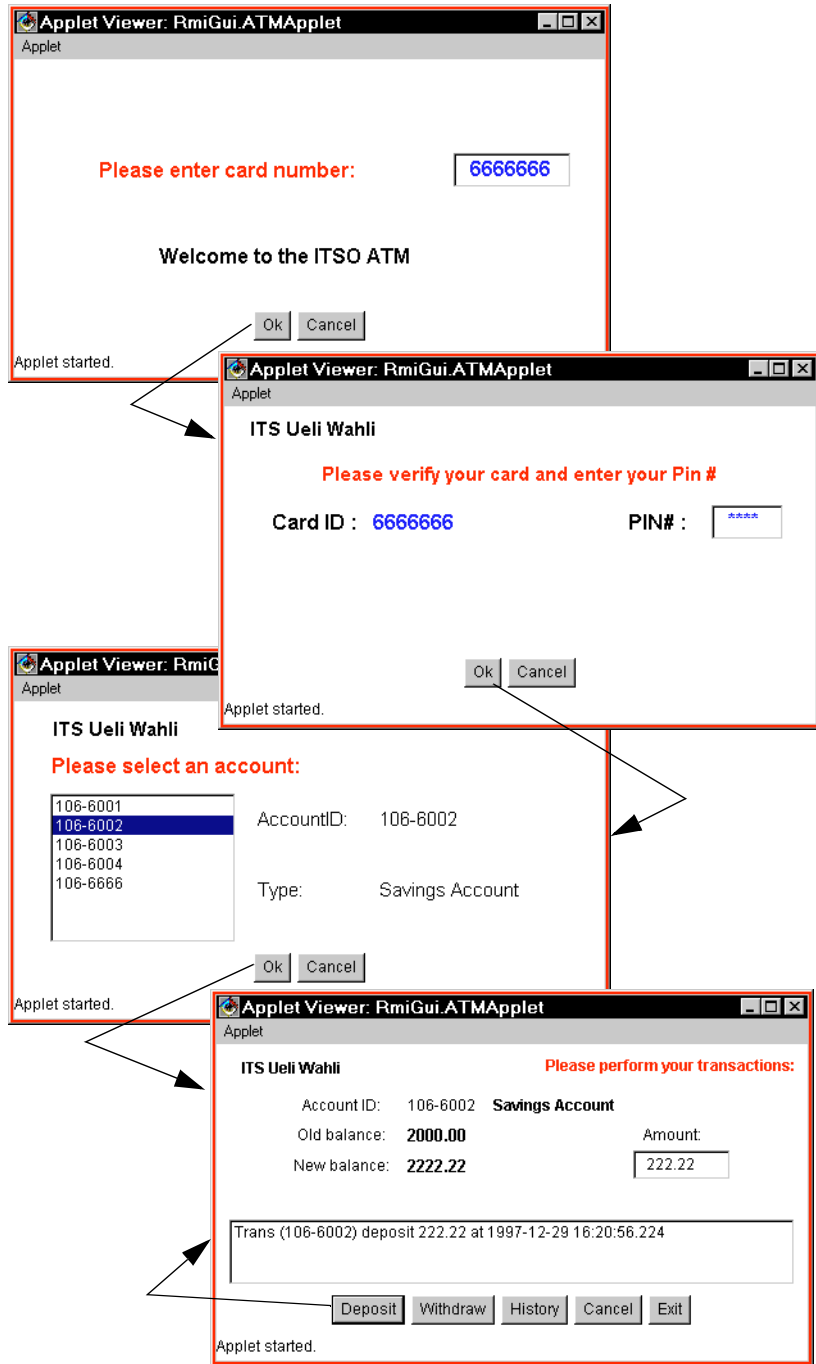
- ❑ The database connection fails. Be sure to test the `AtmDB` applet, because it connects to the database.
- ❑ A switch to the next panel does not occur. Check that the next method of the layout manager is connected and that the condition that triggers the next method actually occurs.
- ❑ Another problem that could occur is related to the sequence in which the beans are initialized through the connections. If you face such a problem, you often can resolve it by reviewing the sequence in the *Reorder Connections From* window of the critical bean. This does not help for events of different origins. You might want to look into the `VisualAge` generated methods, such as `init`, `initConnections`, and the getter method of the critical bean. These methods enable you to add user code indicated by these comment lines:

```
// user code begin
// user code end
```

By writing user code between the two comment lines you can overrule the code sequence generated according to `VisualAge`'s internal algorithm.

- ❑ To catch all exceptions for testing, be sure to activate the commented lines in the `handleException` method of the applet, the subpanels, and data access classes.

After a successful test, we suggest that you version the classes.

**Figure 137.** Sample Run of ATMApplet

Distributed ATM Application

The original master plan was to distribute the ATM application and run the controller with all database accesses on a server machine. This might seem difficult but, if the design is right, it should be rather smooth.

Actually, this is the easy part. Once we have a well-structured application with crisp interfaces, adding RMI function through the RMI Access Builder of VisualAge for Java Enterprise does not require much effort. We now guide you through the few steps necessary.

We copied the existing GUI classes into a new package named `RmiGuiD` and renamed the applet to `AtmRmiApplet`.

Application Changes

Before attempting to run the RMI Access Builder, we have to make a few changes to the beans and the application.

Make the Beans Serializable

First we have to ensure that our beans can be serialized to be transmitted over the network. Every class that is a parameter or result in a call to the controller must be serializable. To make a class (bean) serializable, we add to the class definition:

```
implements java.io.Serializable
```

We add this to all business objects, that is, `Card`, `Customer`, `BankAccount`, `CheckingAccount`, `SavingsAccount`, and `Transaction`. Because all the properties of our beans are either basic data types or classes that have the `Serializable` interface granted from Java or are references to our business objects, this is all we have to do as far as serialization is concerned.

Mark the Methods That Update the Bank Account As Synchronized

Because the server bean can be invoked concurrently from multiple clients, we mark all the methods of the `ATMApplicationController` with the `synchronized` keyword, so that only one client can update an account at any time. For example, we add the `synchronized` keyword to the `deposit` method:

```
public synchronized BankAccount deposit(BankAccount account, ...)
```

Review the Events

In a distributed environment, we have to look at three ways in which events are handled:

❑ Events with emitter and listener on the same system

If the event is both fired and listened to on either the client or the server side, nothing has to be done; it just works the same as before distribution. Examples of this are the `pinCheckedOk` and `pinCheckNotOk` events the `Card` fires depending on whether the check of the PIN was successful or not.

❑ Events fired by an RMI server bean

In this case, the RMI Access Builder generates the necessary classes, methods and interfaces to handle the events, and to ensure that events fired by the server bean are propagated to the client proxy and fired there as well.

In our application, the `cardNotFound` and `DBOutOfSynch` events of the controller bean are examples of this kind of event.

❑ Events with emitter and listener on different systems

These are the events to watch for, because they require some intervention. We have such an event in our application: the `limitExceeded` event that is fired by the `BankAccount` bean when a withdraw transaction is rejected due to a violation of the limit set for that account.

Let us look first at what happens in this case:

- The `withdraw` method of the controller proxy bean is called in the applet, with the account, amount, and card as parameters.
- The execution of the method is delegated to the server bean, and the parameters are passed to the server. Because RMI passes the parameters as copy, and not as reference, we now have a clone of the account bean on the server.
- The actual `withdraw` method is performed against this clone on the server, and on completion, the account is passed back and the state information in the originating account bean is updated.
- If, during the processing on the server side, the bank account fires the `limitExceeded` event, a listener is not registered to handle this event, and, therefore, the account bean in the applet never fires the event.

The way we get around this problem is by propagating the event to the controller, that is, the controller becomes a listener, and when the event arrives, the controller fires an event of its own.

Note: We already implemented the `limitExceeded` event in the design of the controller (see “Event Propagation” on page 199).

Create the Proxy Beans

The `AtmApplicationController` bean is responsible for connecting the layers or subsystems of our application. Therefore, it is only for this bean that we have to invoke the RMI Access Builder.

We select the bean and *Tools -> Remote Bean Access -> Generate Proxy Beans* to get the SmartGuide (Figure 138). We enter `AtmDistributedController` as the proxy bean name and let the RMI Access Builder do its job.

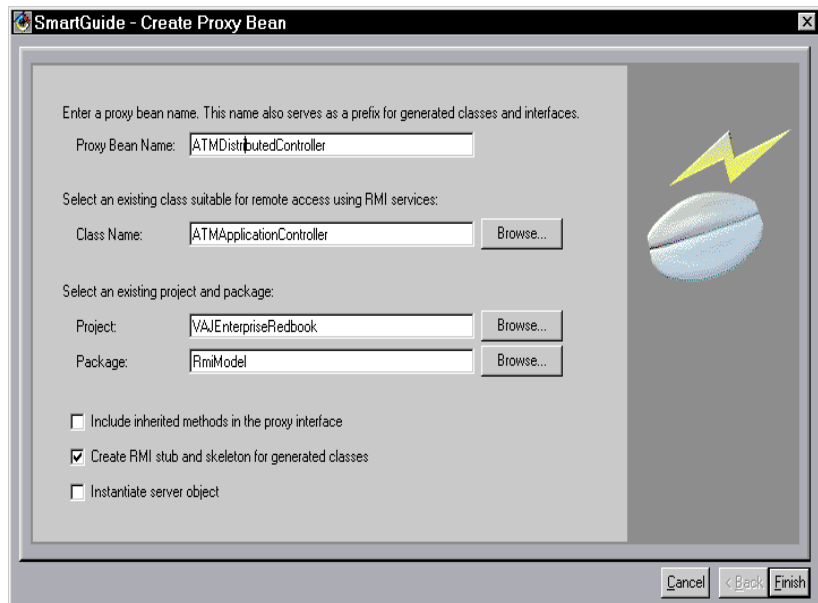


Figure 138. Creating the `ATMDistributedController` Proxy Bean

If we look at the generated beans we find:

- ☐ `AtmDistributedController`, the client proxy
- ☐ `AtmDistributedControllerS`, the RMI server bean
- ☐ The client side skeletons and stubs
- ☐ The server side skeletons and stubs

- ❑ A new listener interface for all events that the `AtmApplicationController` fires or is a listener for.
- ❑ Event classes for all events that the `AtmApplicationController` fires or is a listener for.

These beans are not meant to be changed. If you have to change the application for whatever reason, make the changes in the original beans and regenerate the RMI beans.

All that is left to do is to adjust the applet so that it accesses the server through the client proxy.

Modify the GUI

Changing the ATM applet GUI to use the proxy bean instead of the original controller is not much work. Basically we have to replace the `AtmApplicationController` bean in all panels with the generated proxy bean, the `AtmDistributedController`, and make a few changes in the subpanels.

Using the Distributed Controller

Figure 139 shows the composition of the distributed applet.

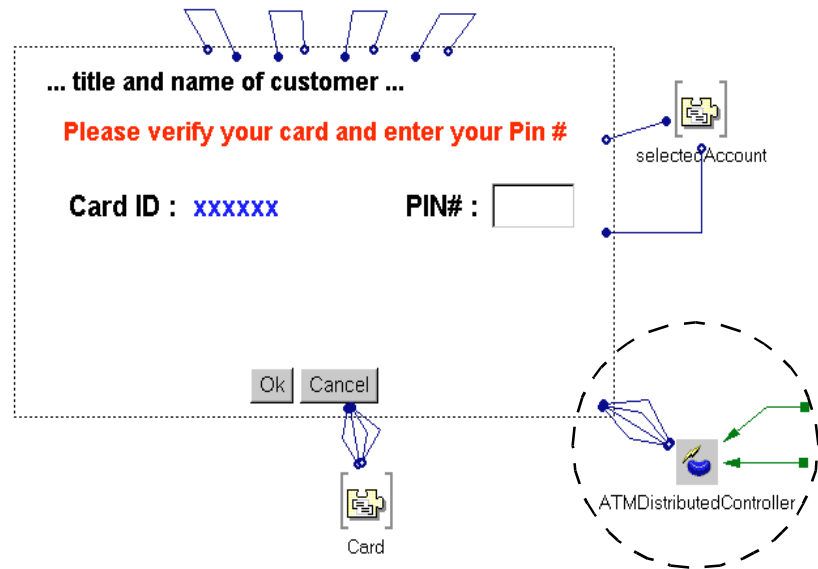


Figure 139. Visual Composition of Distributed ATM Applet

As you can see, there is only one change; instead of the old controller bean we have the `AtmDistributedController`. We follow these steps to make the changes:

- ❑ We drop the new controller bean on the free-form surface.
- ❑ We drag all connections from the `AtmApplicationController` to the `AtmDistributedController` and delete the old controller.
- ❑ We open the properties of the `AtmDistributedController` bean:
 - We set the `parentComponent` property to the value *this*.
 - We set the host name and the port number properties according to the settings of the RMI server; in our case, *localhost* for the host name, and *1099* (or *-1*) for the port. We also set the `trace` property to *true* for testing.
- ❑ In the four subpanels we change the type of the controller bean from `AtmApplicationController` to `AtmDistributedController`.

Changes in Subpanels

Some changes are necessary in the subpanels to deal with the results of the remote methods.

In the PIN panel, the `getAccounts` method returns the modified `Card` object. This object replaces the current `Card` object and stops the firing of further connections for the `pinCheckedOk` event because the source object (the `Card`) changes. The solution is to attach the switching of subpanels to the `normalResult` of the `getAccounts` method (Figure 140).

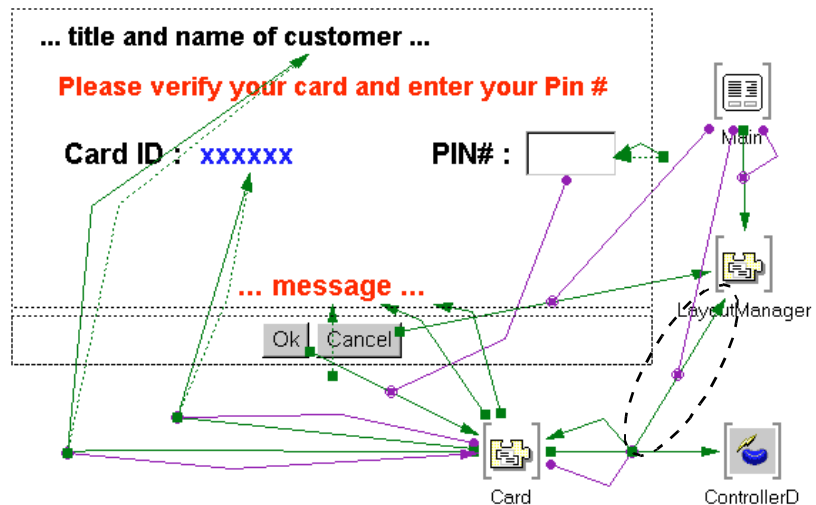


Figure 140. Visual Composition of the RMI PIN Panel

A similar problem exists in the TransactionPanel. The deposit, withdraw, and getTransaction methods of the controller return the modified BankAccount.

In the local solution we could disregard the result because all of the changes occurred in the local BankAccount object. In the RMI solution we have to capture the resulting BankAccount object and modify the local object.

We could replace the local BankAccount object with the result of the methods, but that creates a few problems. The first problem is that the property events for the balances and transactions are not fired, and the second is that the original account object in the accounts vector of the card is not updated.

In our solution we create a new local BankAccount object in a variable and then propagate the changed properties to the original local BankAccount (Figure 141).

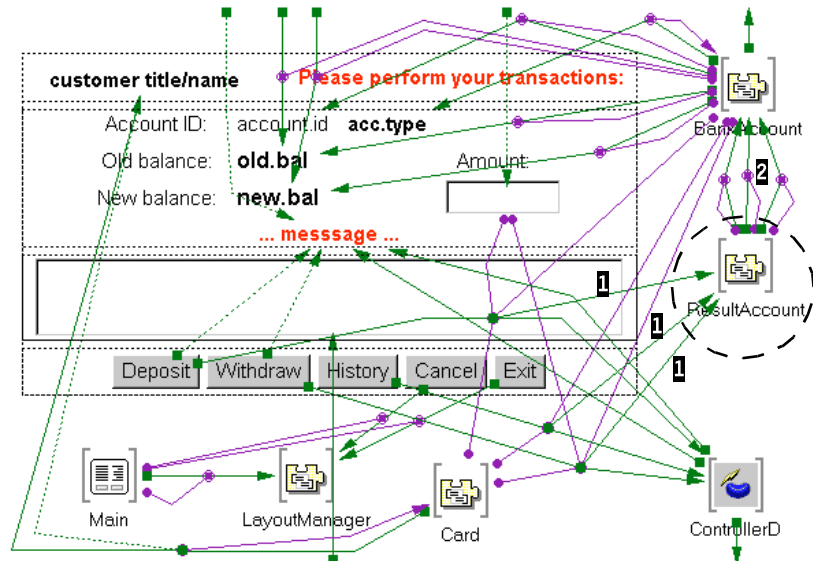


Figure 141. Visual Composition of the RMI Transaction Panel

The normalResult of the deposit, withdraw, and getTransaction method invocations is assigned to the this property of a new ResultAccount variable of type BankAccount (1).

The this event of the ResultAccount variable is connected to the balance, oldBalance, and transaction properties of the BankAccount variable, passing the properties as parameters (2). These connections propagate the changes to the panel fields and into the accounts vector of the Card object.

Test the Distributed ATM Application

First we must ensure that the RMI registry is running. Then we instantiate the RMI server bean, `AtmDistributedControllerS`, in the Remote Object Instance Manager.

Now we can start the `ATMRmiApplet`. The behavior should be the same as in our stand-alone version. In the console window we can follow the trace and check whether it works as required or just as programmed.

There is, however, a common pitfall. The “Security Manager already set” error message in the console window indicates that the `parentComponent` property of the client proxy bean (`AtmDistributedController`) is not set. Make sure that it is set to *this* (the applet itself), or that a connection is made from the applet to the property, and that the connection is fired before the call to the `connect` method of the controller. You can edit the `initConnections` method and copy the connection call into the first hook-up for user code. After reloading the applet, the problem should be gone.

Running the Applet on a Client

If we want to run the applet on a real client machine, we have to set the IP server name in the `AtmDistributedController` bean. We can set the server name in the `initConnections` method at the very start:

```
private void initConnections() {
    // user code begin {1}
    String hostname = "";
    try { hostname = this.getCodeBase().getHost(); }
    catch (Exception e) { }
    if ( !hostname.trim().equals("") )
        getATMDistributedController().set__serverName(hostname);
    // user code end
    ....
}
```

Because an applet can only connect to the server from which it came, we use the `getCodeBase` method of the applet to get the URL of the server from which the applet was loaded. The `getHost` method returns the host name that we use to set the server in the controller bean. When testing under VisualAge for Java on the same machine, the host name is returned as an empty string.

Running As an Application

When we created the applet using SmartGuide, we made sure that we could also run the applet as an application.

An application can run on a client and connect to any server. One way of passing the server TCP/IP host name to the application is through a parameter of the main method. We edit the main method and add the call to the `AtmDistributedController` to set the host name:

```
public static void main(java.lang.String[] args) {
    ATMRmiApplet applet = new ATMRmiApplet();
    java.awt.Frame frame = new java.awt.Frame("Applet");
    frame.addWindowListener(applet);
    frame.add("Center", applet);
    frame.resize(350, 250);
    frame.show();
    applet.init();
    if (args.length == 1)
        applet.getATMDistributedController().set__serverName(args[0]);
    applet.start();
}
```

We start the application with this command:

```
java ATMRmiApplet servername
```


8

Host CICS Access with the CICS Access Builder

The CICS Access Builder is the IBM VisualAge for Java feature that generates classes to interact through the CICS Gateway for Java with an MVS system running CICS. In this chapter we discuss the CICS Access Builder approach to CICS interaction and explore how our ATM application might work if the underlying database were accessed through a CICS host.

Before you read this chapter, please review “Installation and Setup of CICS Components” on page 362.

Host CICS Access Overview

In this section we look at how Java can interact with an existing host CICS system. You will find that the three-tier model fits such an application naturally. You will also find some constraints for our Java program which are imposed by the host system.

CICS

CICS provides a transaction processing server, often for an MVS host, although you can run a CICS server on a workstation server. A typical CICS transaction starts when a user presses the Enter key at a workstation. The workstation transmits the transaction information to the CICS server, where CICS identifies the application program to process the data. CICS starts the program, which retrieves the transaction data, interacts with some data resource (perhaps an SQL or DL/I database, or a VSAM file), and then transmits some result information back to the workstation. CICS looks after the data resources; the update's synchronization, commit, or rollback; and the allocation and deallocation of the resources to run the application program.

Many existing host applications—so-called *legacy* applications—use CICS. It is quite likely that a new enterprise Java application will need to access legacy data resources handled by CICS.

CICS Gateway for Java and CICS Access Builder

The CICS Gateway for Java is part of CICS; the CICS Access Builder is part of VisualAge for Java. These two components complement each other:

- ❑ The CICS Gateway for Java provides communications and data conversion between Java applications and CICS servers.
- ❑ The CICS Access Builder simplifies the Java programming by encapsulating the CICS transactions and data as Java beans.

The CICS Access Builder provides a very simple way to map resources managed by a CICS enterprise server into Java objects. The Java application can manipulate the data like any other Java objects, and then use the beans to cause CICS to process the manipulated data (Figure 142).

The Gateway converts between Java data and the format needed for CICS programs, starts the CICS transaction with the supplied data, takes the result from CICS, converts it to Java data, and returns that data to the Java application.

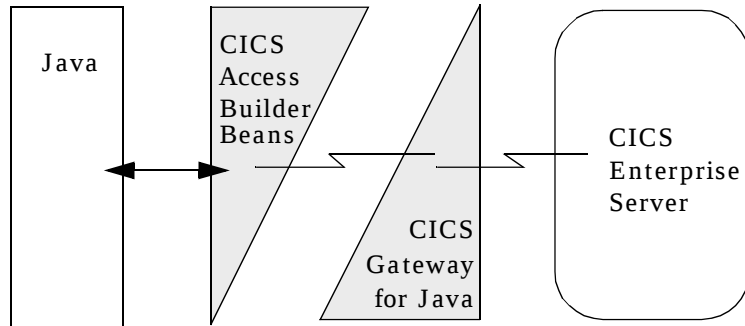


Figure 142. CICS Access Builder Beans and CICS Gateway for Java

How Does the CICS Access Builder Work?

The CICS Access Builder creates a Java bean, which encapsulates the data transferred between the CICS processing program and the Java application, together with classes to handle the data marshaling and conversion. The Access Builder comes with a class library including a bean corresponding to the CICS *unit of work*, for synchronizing with the CICS processing program.

CICS ensures that the unit of work either completes successfully, with all the data updates committed, or else CICS rolls back all of the data updates, so that the underlying data store is never in an inconsistent state.

A Java application using a CICS enterprise server is not concerned with the database—that is the CICS server's responsibility. The application simply collects the data, sends it to CICS for processing, and receives the result from CICS.

This distribution of responsibilities leads to a very clear three-tier architecture, as shown in Figure 143.

In this three-tier architecture, the Java client application is not concerned with the database or data transfer; its sole concern is the user interface and the client application.

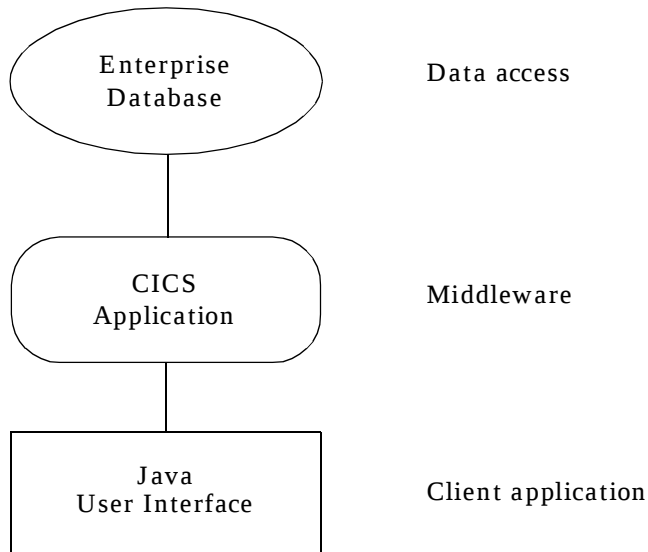


Figure 143. Three-Tier CICS Java Application

Working with a CICS Enterprise Server

It is likely that your enterprise CICS server already manages access to legacy application data and programs. Such resources will have rules for and controls on their use. For example, if there is an SQL database, its definition and contents will be an enterprise resource, so it will be important to conform to the existing access controls.

The three-tier approach lets us comply with such enterprise rules and controls. In particular:

- ❑ We should ensure that any data mapping is handled in the data access layer.
- ❑ Data commit and rollback are handled by the CICS middleware layer.
- ❑ The Java client handles all user interactions.

CICS Java Application Design

Unlike previous examples in this book, we should not be concerned with SQL joins—or indeed with any SQL statements—in Java. They belong in the CICS program or the SQL database itself.

We should also try to ensure that each interaction between the Java application and CICS corresponds naturally to a CICS transaction.

The Java application's interaction is through the bean that maps between the Java and CICS data, and the bean that represents the CICS unit of work. So, for an ideal application, we have the following breakdown of responsibilities:

- ❑ The Java application gets information from the user and sets up the bean to transfer it to CICS.
- ❑ The Java application uses the unit of work bean to transfer the data to the CICS enterprise server.
- ❑ The program that runs as a result in the CICS server performs the business logic to interact with the enterprise database.
- ❑ The enterprise database retrieves the requested data, stores the updates, or carries out whatever processing is appropriate.
- ❑ The CICS program completes the business logic and returns the results to the Java client.
- ❑ The Java client retrieves the results from the CICS data bean and presents them to the user.

CICS Access Builder

VisualAge for Java provides the CICS Access Builder to simplify creating CICS Java applications.

CICS Access Builder: Overview

The following description comes directly from the VisualAge for Java *CICS Access Builder: Overview* help text.

Given the importance and wide-spread use of both Java and the IBM Customer Information Control System (CICS), it is not surprising to find that software developers are looking for ways to enable their Java client programs to remotely access CICS transactions. Unfortunately, accessing a CICS transaction from Java is difficult because CICS programs are often written in COBOL and reside on MVS hosts. The obvious disparity between the Java and COBOL programming languages, plus the differences in internal data representation between Java workstations and MVS hosts, presents a real challenge to the development of any tool that can effectively enable access between Java and CICS.

However, this challenge has recently been met by the development of the CICS Access Builder. This VisualAge for Java tool makes it easy for a Java client program, run as either an applet or as a stand-alone application, to remotely and seamlessly access a CICS transaction. The CICS Access Builder consists of two components: the SmartGuide to create the COMMAREA bean and the run-time class library.

Create COMMAREA Bean SmartGuide

The SmartGuide parses the communications area of a local COBOL file that has been downloaded from an MVS host. The SmartGuide imports the COBOL file and generates a COMMAREA bean and associated classes. The COMMAREA bean and associated classes contain the Java representation of the COBOL communications area, which consists of a group of records that map COBOL data to Java data, one-to-one.

When a client program is run as either an applet or application, the COMMAREA bean is passed as a parameter to the `invokeTxn` or `asynchInvokeTxn` method of the `IVJCicsUOWInterface` bean, the CICS unit of work. These class library methods convert the contents of the COMMAREA bean to COBOL in a form that is acceptable to the MVS host, and it passes the data as part of a request to the IBM CICS Gateway for Java. The CICS Gateway for Java passes the converted data to the CICS Client, which in

turn forwards the data to the CICS region on the MVS host. The CICS region invokes the CICS transaction program to process the data. Once the CICS transaction program processes the data, it sends a reply back to the client program via the CICS Client and the CICS Gateway for Java. The CICS Access Builder run-time class library then converts the COBOL data to Java.

All conversion of the data takes place on the applet or application client, so you must ensure that there is no conversion of the communications area taking place on the CICS server or any intermediate server. The COMMAREA data built by the Java application or applet is correct for targeting COBOL on the host.

Run-Time Class Library

The CICS Access Builder run-time class library provides execution-time support to the COMMAREA bean and other generated classes. The class library hides the complexity of the generated code and provides the generated classes with some generic services.

Of the numerous run-time classes provided with the CICS Access Builder, the IVJCicsUOWInterface class is particularly important. This class contains `invokeTxn` and `asynchInvokeTxn` methods that accept the COMMAREA bean as a parameter and synchronously or asynchronously invoke a CICS transaction. The IVJCicsUOWInterface class also contains methods which are used to manage logical units of work:

startUOW	Starts a unit of work
endUOW	Ends a unit of work
commitUOW	Commits a unit of work
backoutUOW	Rolls back a unit of work

The following scenario helps to clarify the relationship between these four methods of the IVJCicsUOWInterface class:

- ❑ A unit of work is started by calling the `startUOW` method.
- ❑ One or more transactions are invoked using the `invokeTxn` or `asynchInvokeTxn` method.
- ❑ If a transaction does not run correctly, as evidenced by incorrect results or a return code, the unit of work can be backed out by using the `backoutUOW` method. Once the `backoutUOW` method has completed, any changes that were made as a result of invoking transactions are undone.

- ❑ If all transactions run correctly and no problems are encountered, the unit of work can be committed using the commitUOW method.

Once the commitUOW method has completed, any changes that were made as a result of invoking transactions will become permanent. An alternative to using the commitUOW method is to use the endUOW method. The endUOW method is used to send the commit along with the last transaction invocation request. This is accomplished by calling the endUOW method on the IVJCicsUOWInterface class prior to the last invokeTxn request for the unit of work. This causes the last invokeTxn method to do an implicit commit.

A unit of work cannot span operations directed to different hosts and only one request in a unit of work can be outstanding at any given time. For this reason, asynchronous requests should be used with caution.

The IVJCicsUOWInterface class also contains methods which are used to add and remove listeners for the following events:

- ❑ The invocation of a CICS transaction
- ❑ The starting of a unit of work
- ❑ The ending of a unit of work
- ❑ The committing of a unit of work
- ❑ The roll back of a unit of work
- ❑ The occurrence of an exception

ATM Application with the CICS Access Builder

We encounter some ambiguity when we talk about our ATM application with its credit and debit transactions, and our CICS logical units of work corresponding to CICS transactions. In this section, we use *transaction* when we talk about the ATM application and the term *interaction* when we talk about the CICS program.

If we look at the ATM application data flow, we can see the following interactions triggered each time the user presses the Enter key (or an appropriate push button):

- ❑ Send the entered card ID to CICS and retrieve the customer title and name.
- ❑ Send the stored card ID and the entered PIN to CICS for validation, retrieving the list of account IDs for this card (assuming the PIN is valid).
- ❑ Send the card ID, PIN, and selected account ID to CICS and retrieve the account type and balance.
- ❑ Send the card ID, PIN, account ID, current balance, and transaction type and amount to CICS and retrieve the new balance (assuming the transaction is valid).
- ❑ Send the card ID, PIN, and account ID to CICS and retrieve the transaction history.

There are probably two surprises in this list:

- ❑ We send the card ID and PIN with every interaction after the initial logon. That is a normal part of security—we want to ensure that each interaction is properly authorized. The CICS program serves hundreds or thousands of ATMs, so it cannot remember which ATMs have valid cards in them.
- ❑ We send the current balance with the transaction. Thus the CICS program can check that there have been no changes affecting this account from other banking systems. We do not want to use SQL record locking, because the CICS program wants to commit each unit of work as soon as it returns data to the Java application; the CICS program has lots of other ATMs to serve.

There is one other problem to consider. The CICS program returns a list of items in two circumstances: the list of account IDs and the history list of transactions. The CICS data mapping bean does not allow the transfer of unlimited lists. We have to truncate the lists at some point. There are two possibilities:

- ❑ The business imposes a limit on the number of accounts covered by one card and a limit on the number of transactions listed in the history

or

- ❑ We run the retrieve account numbers and retrieve transaction history interactions in a loop until we have retrieved all records. Because the CICS program cannot remember how much it has transferred to us already (remember, it has many other ATMs to serve), we have to send it the highest account ID or transaction ID retrieved so far.

For our sample ATM application we choose the first option and impose a limit of up to 10 accounts per card and the last 20 transactions listed in the history.

CICS Program Design

From the preceding discussion, we can see that the CICS program's responsibilities can thus be divided:

- ❑ Given the card ID, return the customer information
- ❑ Given the card ID and PIN, return the account numbers
- ❑ Given the card ID, PIN, and account ID, return the current balance and account type
- ❑ Given the card ID, PIN, account ID, current balance, and deposit or withdraw transaction, process the transaction and return the new balance
- ❑ Given the card ID, PIN, and account ID, return the transaction history

It is easy to treat these responsibilities as separate CICS programs. Such an approach ensures that the less frequently used programs (such as the transaction history) can be released from the CICS system if they are not in use.

We have ignored error conditions such as an invalid PIN in the list of responsibilities; we will have to return an error indicator for each program. We assume that we can send a clear PIN between the Java client and the CICS server, and that this is sufficient for security checking. Of course, for a real ATM system, we would need a much more sophisticated security system that uses cryptographic protocols.

Accessing the Database

In this example, the CICS programs access an SQL database. We need the following views and SQL statements for the programs described in the preceding discussion:

- ❑ For the first interaction, to get the customer information for a given card:

```
SELECT C.title, C.fname, C.lname
  INTO :title, :fname, :lname
  FROM Card K, Account A, Customer C
 WHERE K.cardid = :cardid
    AND A.cardid = K.cardid
    AND C.custid = a.custid;
```

- ❑ To get the account numbers for a given card and PIN:

```
SELECT A.accid
  FROM Card K, Account A
 WHERE K.cardid = :cardid
    AND K.pin = :pin
    AND A.cardid = K.cardid;
```

- ❑ To get the account type and balance for a chosen account ID:

```
SELECT A.acctype, A.balance
  INTO :acctype, :balance
  FROM Card K, Account A
 WHERE K.cardid = :cardid
    AND K.pin = :pin
    AND A.cardid = K.cardid
    AND A.accid = :accid;
```

- ❑ To process a deposit or withdraw transaction:

- Validate card ID, PIN, and account ID, getting the account type and balance (to check that it is unchanged):

```
SELECT A.acctype, A.balance
  INTO :newacctype, :newbalance
  FROM Card K, Account A
 WHERE K.cardid = :cardid
    AND K.pin = :pin
    AND A.cardid = K.cardid
    AND A.accid = :accid;
```

- Update the account transaction history:

```
INSERT INTO Trans(Transid, Accid, Transtype, Transamt)
VALUES (Current Timestamp, :accid, :transtype, :transamt);
```

- Update the account balance:

```
UPDATE Account
  SET Balance = :balance
 WHERE Accid = :accid;
```

❑ To retrieve the transaction history:

```
SELECT Date(T.transid), Time(T.transid), T.transtype, T.transamt
FROM Card K, Account A, Trans T
WHERE K.cardid = :cardid
      AND K.pin = :pin
      AND A.cardid = k.cardid
      AND A.accid = :accid
      AND T.accid = a.accid
ORDER BY T.transid DESC;
```

Building the CICS Programs

CICS programs communicate through the CICS COMMAREA. The CICS Gateway for Java and the CICS Access Builder handle the content conversion, and the CICS unit of work bean handles the program scheduling and commit and rollback.

The programs must be written in a language supported by the CICS program translator (which expands the EXEC CICS statements into calls to CICS services, and so forth). COBOL has a particular advantage, because the CICS Access Builder handles COBOL COMMAREA definitions directly. If we use another language, such as PL/I, we must translate the COMMAREA definition between COBOL and PL/I ourselves. We choose COBOL programs here but also show what is involved for the PL/I (or other language) case.

COBOL Input to the CICS Access Builder

The Access Builder builds the COMMAREA bean and its supporting classes by finding the COMMAREA definition in a COBOL program.

It expects to process the source code of a file containing a syntactically correct COMMAREA structure declaration alone, or a syntactically correct enterprise system COBOL program.

For example, it accepts the short code shown in Figure 144 or the full program code shown in Figure 145.

<pre>01 DFHCOMMAREA. 02 name PIC X(20) DISPLAY. 02 persno PIC 99999 DISPLAY. 02 salary PIC \$99999 DISPLAY. 02 message PIC X(64) DISPLAY.</pre>

Figure 144. Sample CICS COMMAREA Structure Definition

```

identification division.
program-id. CICSSAMP.
environment division.
data division.
working-storage section.
01 tmp pic a(40).
LINKAGE SECTION.
01 DFHCOMMAREA.
   02 name      PIC X(20)  DISPLAY.
   02 persno    PIC 99999  DISPLAY.
   02 salary    PIC $99999 DISPLAY.
   02 message   PIC X(64)  DISPLAY.
procedure division.
start-para.
   move 'Transaction finished' to message.
   move 'ADDER transaction executed.' to tmp.
   EXEC CICS WRITE OPERATOR TEXT(tmp) TEXTLENGTH(27)
   ACTION(2) END-EXEC.
   EXEC CICS RETURN
   END-EXEC.

```

Figure 145. Sample CICS Program Including COMMAREA Definition

If the input is not a valid COBOL program or uses syntax extensions specific to a workstation COBOL compiler, the Access Builder program stops, usually (but not always) with an error message. As an example of something it will not process, a copy instruction for workstation COBOL might read:

```
copy "sqlca.cbl".
```

compared with the mainframe version without quotes and file extension:

```
copy sqlca.
```

The CICS Access Builder does not accept EXEC SQL ... statements before the COMMAREA declaration, so if your CICS program uses SQL resources, you may have to use the COBOL output from the SQL preprocessor, rather than the original COBOL source input to it.

Restrictions

There are several restrictions on the field definitions permitted in the COMMAREA. These are documented in the VisualAge for Java online help. The most important restrictions are:

- ☐ COBOL level-88 records and REDEFINES clauses are ignored.
- ☐ COBOL POINTER formats are not supported.
- ☐ COBOL OCCURS is supported (mapped to an array), but OCCURS DEPENDING ON is not.
- ☐ COBOL copybook statements are not supported within the COMMAREA.
- ☐ COBOL 85 object-oriented extensions are not supported.
- ☐ The code page in use at the enterprise system host should be IBM-037, storing integer values in “Big-endian” format. You get incorrect conversion if you communicate with a workstation CICS system (such as Transaction Server for OS/2) instead of an enterprise CICS server. DBCS (graphical) character sets are not supported.
- ☐ There are no set or get methods for arrays; you must update or retrieve array elements manually, using assignment statements.
- ☐ COBOL level-66 and level-77 records terminate the COMMAREA definition.
- ☐ The CICS Access Builder does not support character variables longer than 249 bytes. (This is a bug.)

Data Types and Non-COBOL Programs

The CICS Access Builder generates classes converting between the data representations in the CICS system and Java. Table 19 shows the conversions provided in VisualAge for Java Version 1.0.

Table 19 also includes representations in SQL, C, and PL/I for the common COBOL data types. This gives you information on how to code a dummy COBOL program and COMMAREA equivalent to a COMMAREA in a program in one of these other languages.

Table 19. Data Types Supported by the CICS Access Builder

COBOL	Java	SQL	C	PL/I
pic 9(4) usage comp	short	smallint	short int	fixed bin(15)
pic S9(4) usage comp (or wider)	int	integer	long int	fixed bin(31)
pic 9(4) usage display	short			pic'9(4)'
pic S9(4)	int			pic'S9(4)'
pic S9(5) usage comp-3	int	decimal(5)	decimal(5)	fixed dec(5)
pic S9(15) usage comp-3	long	decimal(15)	decimal(15)	fixed dec(15)
pic S999V99 usage comp-3	double	decimal(5,2)	decimal(5,2)	fixed dec(5,2)
pic S999V99	double			pic'S999V99'
usage comp-1	float	real	float	float bin(21)
usage comp-2	double	double	double	float bin(53)
pic x(1)	String	char(1)	char	char
pic x(249)	String	char(249)	char[250]	char(249)
	String	varchar(8)	char var[8]	char(8) var
pic x(10)	String	date	char[11]	char(10)
pic x(8)	String	time	char[9]	char(8)
pic x(26)	String	timestamp	char[27]	char(26)

All forms of character strings map to Java strings. COBOL has no data type for varying length strings, so mapping to COBOL results in fixed-length strings.

Several conversions change from short to int or long, depending on the width of the COBOL data type. The widths in the table are the most typical values.

Coded Decimal Data

If you need to transfer and manipulate financial data stored in fixed-point decimal encoding in CICS, note that the CICS Access Builder generates Java double-length floating-point numbers, not `BigDecimal` values (unlike the Enterprise Access Builder for Data). Therefore you may encounter rounding and truncation effects in floating-point arithmetic results in any Java code you write.

If your application manipulates such financial data, you may find it easier to convert the data from floating point to `BigDecimal` after retrieving it from CICS and then convert back again when the calculations have completed.

If you pass character string data (*COBOL usage display*) rather than fixed-point decimal data (*usage comp-3*), you can convert the data by using `BigDecimal(string)`, which is more predictable and accurate than `BigDecimal(double)`. If you use *usage comp-3*, the marshaling methods convert to double precision floating point, and you lose information about the intended number of decimal places. For example, 12.50 decimal becomes 1.25E1.

Dates, Times, and Time Stamps

COBOL has no special data types for dates and times, so they map to appropriate character strings or numbers. Table 19 includes the COBOL string lengths for SQL dates, times, and time stamps; other encoding, such as *yyyy-mm-dd hh:mm:ss*, are possible. Whatever format you encounter, if you need to do date or time calculations, you have to write Java code to convert the strings to dates or times.

Dummy COBOL Programs

If your enterprise system host programs are written in a language other than COBOL, you have to convert your `COMMAREA` definition to COBOL, using the target data types listed in Table 19.

Figure 146 shows a COBOL declaration for most of the data types in Table 19.


```

01 DFHCOMMAREA.
05 VerySmall      pic S9(3)          usage comp.
05 UnsignedSmall  pic 9(4)           usage comp.
05 SmallInt       pic S9(4)          usage comp.
05 Integer        pic S9(9)          usage comp.
05 UnsignedLarge  pic 9(9)           usage comp.
05 SmallUSString  pic 9(4).
05 SmallIntString pic S9(4).
05 IntString      pic S9(9).
05 Decimal-5-0    pic S9(5)          usage comp-3.
05 Decimal-5-2    pic S9(3)V99       usage comp-3.
05 Decimal-15     pic S9(15)         usage comp-3.
05 Pict           pic S999V99.
05 Real-4         usage comp-1.
05 Real-8         usage comp-2.
05 Char-1         pic X.
05 Char-249       pic X(249).
05 VarChar-8.
10 Len           pic 9(4)            usage comp.
10 Tex           pic X(8).

```

Figure 146. COBOL Example of Most Data Types Supported

Running the CICS Access Builder

We can run the CICS Access Builder from either the Workbench or from the command line. We use the COMMAREA of the sample program in “Sample COBOL CICS Transaction” on page 244:

```

01 DFHCOMMAREA.
05 cardID         pic x(7).
05 PIN            pic x(4).
05 PIN-OK        pic x(1).
05 accounts       pic s9(4).
05 accountInfo    occurs 10 times.
10 account        pic x(8).

```

Running the CICS Access Builder from the Workbench

We normally run the CICS Access Builder from the VisualAge for Java Workbench so that the resulting beans are imported and compiled automatically into a package.

We create a new package called `AtmCICSAccess`. (The builder does not work on the IBM-supplied packages.) We select the *Selected* menu, or we use the pop-up menu on the package and select *Tools -> Host CICS Access -> Create COMMAREA Bean...* as shown in Figure 147.

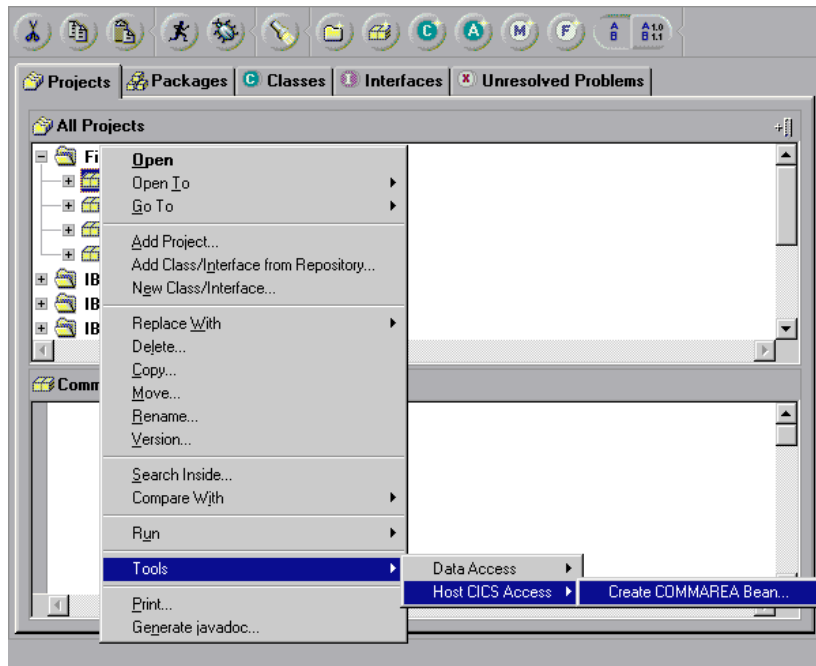


Figure 147. Starting to Create the CICS COMMAREA Bean

This brings up the SmartGuide of the CICS Access Builder (Figure 148), appropriately completed for the COBOL program discussed in “Sample COBOL CICS Transaction” on page 244.

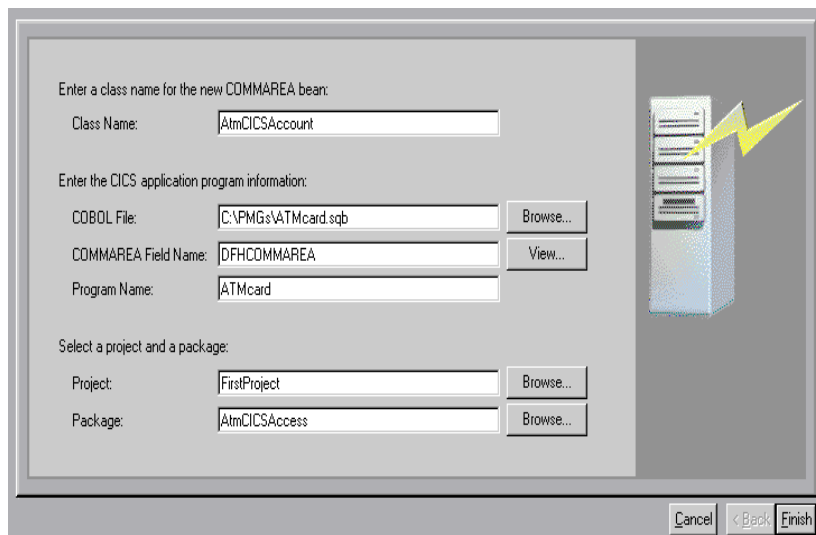


Figure 148. Completing the COMMAREA Bean SmartGuide

After clicking on **Finish** in the SmartGuide, we have new classes created to access the COMMAREA data (Figure 149). Note the distinguishing icons at the end of each generated class.

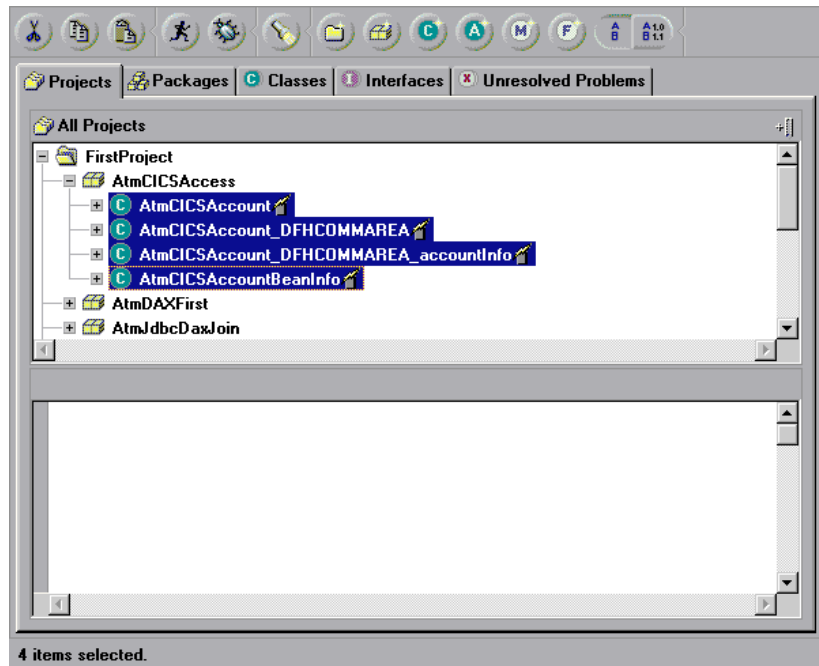


Figure 149. The Result of Creating a COMMAREA Bean

Running the CICS Access Builder from the Command Line

Usually, when you have the source for a correct COBOL program to work with, it is very easy to use the CICS Access Builder from the Workbench.

If you are working with a particularly complex program written in COBOL or some other language, you might need to run the CICS Access Builder several times while you check your syntax or modify code until the Builder accepts it. In this case, it may be easier to run the Builder from the command line.

The name of the CICS Access Builder is `ivjdcics`; it takes as input the COBOL source and creates as output the Java source code for the COMMAREA bean and its supporting classes.

The command line example shown in Figure 150 runs the CICS Access Builder for the same COBOL source as the SmartGuide example; the created Java source is stored in a named subdirectory (`AtmCICSAccess`) of the originating path.

If your program source is in error, the CICS Access Builder returns an error message in the command line output stream; otherwise it displays “Import completed successfully!”

```
D:\CICSTEST>ivjdcics
IVJ5500E: Usage: C:\IBM\JAVA\EAB\BIN\IVJDCICS.EXE
               <Target Path>
               <Package>
               <Class Name>
               <COMMAREA Field Name>
               <Program Name>
               <COBOL File>

D:\CICSTEST>ivjdcics . AtmCICSAccess AtmCICSAccount DFHCOMMAREA
                  ATMcard ATMcard.sqb

IVJ5504I: C:\IBM\JAVA\EAB\BIN\IVJDCICS.EXE:
                  Import completed successfully!
```

Figure 150. Running the CICS Access Builder from the Command Line

The parameters of the command line program are:

- ☐ **Target Path** - the path leading to (but not including) the directory for the package that will contain the generated java code. (Default is the current directory.)
- ☐ **Package** - the name of the package (and therefore a directory name) that will contain the generated code
- ☐ **Class Name** - the prefix used when building the CICS Access Builder bean and conversion and marshaling classes
- ☐ **COMMAREA Field Name** - the name of a CICS structure (or substructure) that maps the COMMAREA to be exchanged with the Java program. Note that this need not be a 01 name, though it often is. It does not have to be the actual COMMAREA; it may be another structure to which the CICS program moves the passed COMMAREA data.
- ☐ **Program Name** - the name of the application program that will be run in the CICS host. This name must be the correct name for the program resource and defined to the CICS system through the CEDA DEFINE PROGRAM command.
- ☐ **COBOL File** - the source file name

The program's messages are written to the standard output stream. The generated Java source code is written to the named package directory stored in the named path.

Generated Classes

This example generates four classes. The main class is the `AtmCICSAccount` bean and its `BeanInfo` class. The bean class contains the getter and setter methods for the basic data types and the event triggering.

The two helper classes `marshal` and `unmarshal` the data exchanged with CICS through the `COMMAREA`. We get one class for each substructure; one for the `COMMAREA` itself, `AtmCICSAccount_DFHCMMAREA`, and one for the `accountInfo` substructure, `AtmCICSAccount_DFHCMMAREA_accountInfo`. We do not usually use these classes in our application code; we only access the main bean.

Note the naming convention for the generated properties. An underscore is appended to the COBOL name (for example, `cardID_`) and a dash is converted to two underscores (`PIN__OK_`). There is no property for the `accountInfo` array.

The generated bean contains a variable, `DFHCMMAREA_`, of type `AtmCICSAccount_DFHCMMAREA`. The actual data variables of the properties are defined in the helper classes.

Application Coding Techniques

For the ATM application we have to deal with exceptions thrown in the CICS programs and with arrays in the `COMMAREA`.

Throwing Exceptions from the CICS Host

We want to throw an exception if the PIN is invalid. However, the CICS host program, not the Java application, validates the PIN. The CICS host program cannot raise a Java exception—it is in the wrong machine. Instead, it sets a `COMMAREA` variable, `PIN-OK`, to 0 or 1 depending on the result of the validation. We have to add Java code in the ATM applet to investigate the matching property, `PIN__OK_`, when we retrieve the data from CICS, and then throw an exception if the value is not 1.

Here is one way to add the event generation to the `COMMAREA` bean. Edit the generated bean and add the two existing PIN validation events on the `BeanInfo` page:

- ☐ Select *New Event Set Feature...* in the *Feature* menu. Enter `RmiModel.PinCheckedOkListener` as the event listener, and `pinCheckedOK` as the event name, then click on **Finish**.
- ☐ Do the same for the `RmiModel.pinCheckedNotOk` event.

- ❑ Update the `setPIN__OK_` method to fire one of the events when the PIN-OK value is set by the CICS transaction:

```
public void setPIN__OK_ ( String aPIN__OK ) {  
    String oldPIN__OK_ = DFHCOMMAREA_.PIN__OK_  
    DFHCOMMAREA_.PIN__OK_ = aPIN__OK_  
    firePropertyChange( "PIN__OK_", oldPIN__OK_, DFHCOMMAREA_.PIN__OK_ );  
    if ( aPIN__OK_.trim().equals("0") )  
        fireHandlePinCheckedNotOk(new RmiModel.PinCheckedNotOkEvent(this));  
    else  
        fireHandlePinCheckedOk(new RmiModel.PinCheckedOkEvent(this));  
    return;  
}
```

Defining the event features enables other beans to register with the COMMAREA bean and handle the events.

Exchanging Array Data with CICS

The host program returns the bank accounts in the `accountInfo` array, and the number of accounts in the `accounts` field.

The CICS Access Builder does not generate getter and setter methods for the array, and we have to access the array elements directly. For example, to create a new `AtmCICSAccount` where the first element of the `accountInfo` is 2222222, we would code:

```
AtmCICSAccount atm = getCOMMAREA();  
atm.DFHCOMMAREA_.accountInfo_[0].account_ = "2222222";  
atm.setAccounts_(1);
```

Note that the `propertyChange` event is not fired automatically for the `accountInfo` array at any point in the `AtmCICSAccount` bean. The `setAccounts_` method fires the `propertyChange` event for the `accounts_` property, which can be used to trigger the appropriate actions.

We can make sure that the `propertyChange` event is fired for the `accounts_` property after the CICS transaction by setting its value to -1 before the call.

We have to be careful with such techniques; although we know that the `accounts_` property contains the number of elements set in the `account_` array, the `AtmCICSAccount` bean does not. Any relationship between the two and with the `accounts_` `propertyChange` event is purely in our code, not the bean's. If we use such a mechanism, we should encapsulate it in getter and setter methods that we code for the `account_` array, as shown in Figure 151.

```
public void setAccount_(int which, String newAccount) {
    if ( which < 0 ) { // New account number, add at end
        which = getAccounts_();
        DFHCOMMAREA_.accountInfo_[which].account_ = newAccount;
        setAccounts_(which+1);
    }
    else
        if ( which < getAccounts_() ) { // replace existing account
            DFHCOMMAREA_.accountInfo_[which].oldAccount_ =
                DFHCOMMAREA_.accountInfo_[which].account_;
            DFHCOMMAREA_.accountInfo_[which].account_ = newAccount;
            // fire property change, number of accounts does not change
            firePropertyChange( "accounts_", -1, getAccounts_());
        }
    return;
}

public String getAccount_(int which) {
    if ( which < getAccounts_() )
        return DFHCOMMAREA_.accountInfo_[which].account_;
    else return null;
}
```

Figure 151. Possible Coding for Array Element Get and Set Methods

Sample COBOL CICS Transaction

For this example (Figures 152 and 153), we look at the transaction where the Java application sends the card ID and PIN to the enterprise CICS, which returns an array of up to 10 account numbers if the PIN is valid, and nothing if the PIN is invalid.

```
*****
** Source File Name = ATMcard.sqb
**
** Called by the CICS Gateway for Java
**
** Arguments are passed and returns in the CICS COMMAREA
** This program retrieves information for a chosen account,
** provided the card ID and PIN are valid for the account.
** Note: this is a hybrid between host COBOL for CICS/ESA
** and workstation COBOL for CICS for OS/2.
** It is probably not correct without modification in either
** environment.
*****
Identification Division.
Program-ID. ATMcard.
Environment Division.
Data Division.
Working-Storage Section.
* Copy Files for Constants and Structures.
  copy sql.
  copy sqlenv.
  copy sqlca.
Linkage Section.
* The DFHCOMMAREA contains the host variables we wish to share
* with SQL. However, the CICS Access Builder does not accept
* EXEC SQL statements, so they are commented out for the run
* of the CICS Access Builder.
*
* EXEC SQL BEGIN DECLARE SECTION END-EXEC.
*
01 DFHCOMMAREA.
  05 cardID          pic x(7).
  05 PIN             pic x(4).
  05 PIN-OK          pic x(1).
  05 accounts        pic s9(4).
  05 accountInfo     occurs 10 times.
    10 account       pic x(8).
  01 prog-name       pic x(12) value "ATMcard".
  01 accID           pic x(8).
*
* EXEC SQL END DECLARE SECTION END-EXEC.
*
77 errloc           pic x(80).
```

Figure 152. COBOL Transaction ATMcard: Declarations


```

Procedure Division.
Main Section.
* Initialise the account info array.
  Move 0 to accounts.
  Move "0" to PIN-OK.
* Declare the cursor for the accounts we identify
  EXEC SQL DECLARE c1 CURSOR FOR
    SELECT A.accID
    FROM card K, account A
    WHERE
      K.cardID=:cardID
      AND K.PIN=:PIN
      AND A.cardID = K.cardID
  END-EXEC.
  move "DECLARE CURSOR" to errloc.
  call "checkerr" using SQLCA errloc.
* Open the cursor
  EXEC SQL OPEN c1 END-EXEC.
  move "OPEN CURSOR" to errloc.
  call "checkerr" using SQLCA errloc.
* Loop to process the cursored selection
  Perform Acct-Loop thru End-Acct-Loop
    until SQLCODE not equal 0.
* If we found no accounts, the cardID and PIN combination
* were invalid.
* A real system would incorporate cryptographic authentication.
*
  If accounts equal 0
    move "0" to PIN-OK.
* Close the cursor and commit the work done.
  EXEC SQL CLOSE C1 END-EXEC.
  Move "CLOSE CURSOR" to errloc.
  Call "checkerr" using SQLCA errloc.
  EXEC SQL COMMIT WORK END-EXEC.
  Move "COMMIT WORK" to errloc.
  Call "checkerr" using SQLCA errloc.
End-Main.
go to End-Prog.
*--- The loop to fetch up to 10 account numbers
Acct-Loop Section.
  EXEC SQL FETCH c1 INTO :accID END-EXEC.
  If SQLSTATE equal SQL-NODATA-EXCEPTION
    go to End-Acct-Loop.
  Move "FETCH" to errloc.
  Call "checkerr" using SQLCA errloc.
* Only take up to 10 account numbers.
  If accounts equal 10
    go to End-Acct-Loop.
* Providing there is space, store this account number
  Add 1 to accounts.
  Move accID to account(accounts).
End-Acct-Loop. exit.
*--- End of the loop to fetch up to 10 account numbers
End-Prog.
exit program.

```

Figure 153. COBOL Transaction ATMcard: Procedure Division

ATM Applet Using Host CICS Access

Now that we have a COMMAREA bean, we can combine it with a CICS unit of work bean, and build an applet. This example models the ATM function where the user sends the card ID and PIN to the server and gets a display of valid account IDs in return.

Visual Composition

The applet is almost entirely a visual construction; it needs an event-to-script connection to propagate the account ID list from the COMMAREA, because the CICS Access Builder provides no properties or methods to access the array of account numbers (Figure 154).

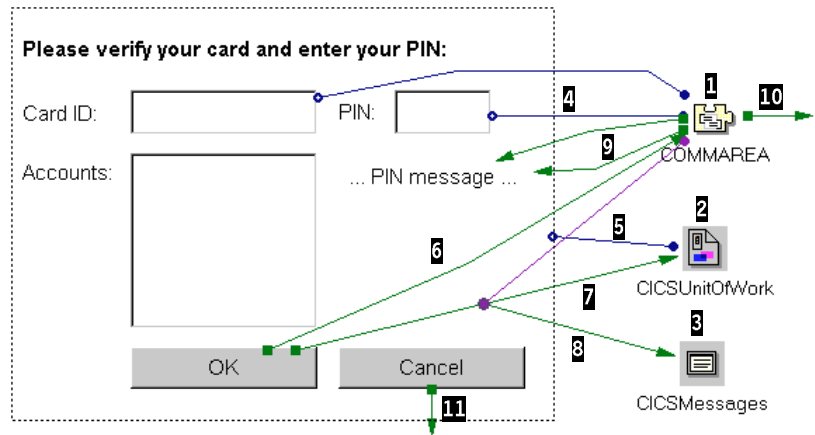


Figure 154. Building the CICS Applet

The applet components are:

- ❑ The COMMAREA bean, of type `AtmCICSAccount` (1).
- ❑ The CICS unit of work bean, called `CICSUnitOfWork` (2). You select the bean from the Enterprise Access palette (the bottom one). You need one CICS unit of work bean for each host system—you cannot share this bean. The bean's properties must be set to conform to your CICS Gateway for Java and CICS host server's configuration. See "Properties of the CICS Unit of Work Bean" on page 248 for our applet.
- ❑ A message box bean, called `CICSMessages`, from the Enterprise Access palette (3). This will produce a pop-up window to display error information from the `CICSUnitOfWork` bean, if necessary.

- ❑ A property connection between the card ID entry field and the COMMAREA bean's cardID_ property, using the textValueChanged event for the source event (4). This ensures that the COMMAREA is updated when the text field changes.
- ❑ A similar property-to-property link for the PIN entry field.
- ❑ A property connection between the applet's this property and the CICSUnitOfWork bean's parent property (5). You need this for the CICSUnitOfWork constructor to work correctly when the Applet opens.

Now we implement the logic of the applet:

- ❑ An event-method connection between the **OK** button and the accounts_ property of the COMMAREA, setting it to -1 (6).
- ❑ An event-method connection between the **OK** button and the CICSUnitOfWork bean's invokeTxn method, passing the COMMAREA as a parameter (7).
- ❑ An event-method connection between the exceptionOccurred event of the transaction invocation with the CICSMessages' showException method, passing the event data as a parameter (8).
- ❑ Two event-method connections to capture the pinCheckedOk and pinCheckedNotOk events of the COMMAREA, displaying an appropriate message in a text label (9).
- ❑ A property-script connection between the COMMAREA's accounts_ property (the number of accounts in the bean) and a new method, setAccounts, in the applet. Pass the event data, that is, the number of accounts. The script in Figure 155 populates the account ID list from the AccountInfo array (10). There are no bean properties for the array elements in the COMMAREA, so the applet needs a script to populate the Account ID list. Figure 155 shows the code of the setAccounts method.

```
private void setAccounts (int accounts ) {
    // Code to set the accounts list from the accounts in the COMMAREA
    // The parameter is the number of accounts from the COMMAREA
    AtmCICSAccount atm = getCOMMAREA();
    List accountList = getAccountList();

    accountList.removeAll();      // clear the account list

    for (int i=0; i<accounts; i++) // Add items from the COMMAREA
        accountList.addItem(atm.DFHCOMMAREA_.accountInfo_[i].account_);
    return;
}
```

Figure 155. Populating a List from a COMMAREA Array

We also add an event-method connection from the **Cancel** button to a new script called `simulateCICS` (11). The `simulateCICS` method can be used to test the logic without having a CICS Gateway and server running (see “Simulating a CICS Server” on page 249).

Properties of the CICS Unit of Work Bean

The CICS unit of work bean's properties control the connection from the applet through the CICS Gateway for Java to the Enterprise system CICS host are shown in Figure 156.

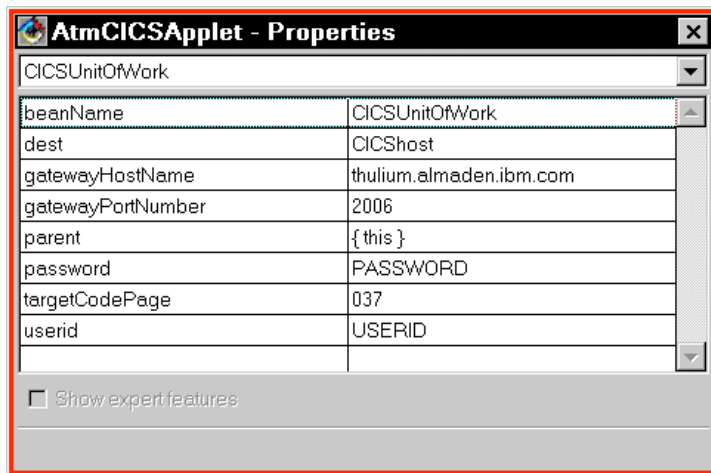


Figure 156. CICS Unit of Work Bean Properties

We must set the properties of the CICS unit of work bean:

- ☐ *dest* is the CICS server destination.
- ☐ *gatewayHostName* is the TCP/IP host name where the CICS Java Gateway runs.
- ☐ *gatewayPortNumber* is the TCP/IP port (default 2006).
- ☐ *parent* is the applet, either *this* or through a connection.
- ☐ *password* is the password for the CICS server transaction.
- ☐ *targetCodePage* is 037 for now; VisualAge for Java 1.0 only supports a CICS MVS server.
- ☐ *userid* is the user ID for the CICS server transaction.

Simulating a CICS Server

If a real CICS server is not available, we can use a small user-written method to simulate the execution of the CICS transaction.

In our case we connect the **Cancel** button to a new script method, `simulateCICS` (Figure 157). This method accepts a few card ID and PIN combinations and fills the `COMMAREA` with account numbers.

```
private void simulateCICS ( ) {
    AtmCICSAccount atm = getCOMMAREA();
    atm.setAccounts_(0);
    if ( atm.getCardID_().equals("1111111") &&
        atm.getPIN_().equals("1111") ) {
        atm.setAccount_(-1, "101-1001");
        atm.setAccount_(-1, "101-1002");
        atm.setAccount_(-1, "101-1003");
        atm.setPIN__OK_("1");
        return;
    }
    if ( atm.getCardID_().equals("2222222") &&
        atm.getPIN_().equals("2222") ) {
        atm.setAccount_(-1, "102-2001");
        atm.setAccount_(-1, "102-2002");
        atm.setPIN__OK_("1");
        return;
    }
    atm.setPIN__OK_("0"); // all others: pin not OK
}
```

Figure 157. Script to Simulate a CICS Transaction

The `simulateCICS` method uses the `setAccount_` method shown in Figure 151 on page 243 to add account numbers. The `setAccount_` method sets the number of accounts, which triggers the execution of the `setAccounts` script to fill the list box with account numbers. Finally the `simulateCICS` method sets the `PIN__OK_` property, which triggers one of the two PIN events.

Simulation lets you test the logic of the application and the propagation of events, such as the property change event of the number of accounts and the PIN events that are fired by PIN validation.

Installing CICS and Java Components

Our Java application needs two components to build the communications path between it and the enterprise CICS system (Figure 158):

- ❑ The CICS Gateway for Java
- ❑ A CICS client—in our case the CICS Client for Windows NT

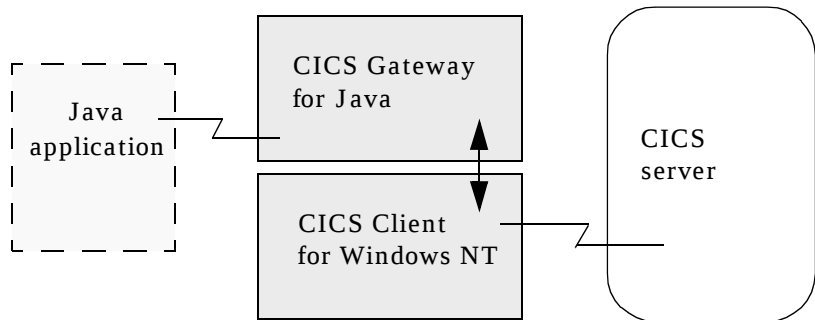


Figure 158. CICS Server, CICS Client, and CICS Java Gateway

The installation of the CICS Client for Windows NT and the CICS Java Gateway and the setup of the components for communication with a CICS server are discussed in “Installation and Setup of CICS Components” on page 362.

Current Restrictions

VisualAge for Java 1.0 supports only a CICS server on MVS. The CICS unit of work bean must be set to use code page 037.

Note that some common browsers do not support code page 037. This restricts the usage of applets with CICS transaction access. The current support is designed for applications installed on client machines or running on middle-tier servers.

CICS Host Access Topologies

Now that we have discussed the components for CICS host access (the VisualAge for Java CICS Access Builder, the CICS Gateway for Java, the CICS Client and the host CICS enterprise system), we can review the connectivity and topology options, and see how they might fit into client/server strategies.

CICS Gateway for Java and CICS Client Topologies

The CICS Gateway for Java comes in two flavors:

- ❑ CICS Gateway for Java for MVS
- ❑ CICS Gateway for Java Workstation

The MVS version allows a Java client to connect across a TCP/IP network to the MVS host, and hence directly to the enterprise CICS system. This direct connection offers simplicity where all Java applications connect to the same server.

The workstation version usually connects to a CICS Client on the same workstation, which in turn connects to the CICS server. The Java clients connect to the Gateway through TCP/IP; the Gateway connects to the CICS server through APPC, TCP62, or TCP/IP, depending on the type of server and the network constraints. This configuration offers flexibility, because the Gateway can provide unchanged connections to the Java clients while the server topology changes. It also provides a parallel topology where the Java clients are loaded from a Web server; the Gateway and the Web server can coexist.

The CICS Gateway for Java can coexist with a CICS for OS/2 server, bypassing the need for a CICS Client on the workstation. This configuration offers the same simplicity as the CICS Gateway for Java for MVS offers with an enterprise system host. However, the CICS Access Builder of VisualAge for Java Version 1.0 does not support CICS for OS/2.

Client/Server Tier Topology

The VisualAge for Java enterprise access tools offer flexibility on a grand scale, so that a Java application can mix CICS, SQL, C++, and RMI access. With the CICS connectivity options, we can have the Gateway for Java on a workstation CICS client or an enterprise CICS server. With this flexibility a complex web of interconnected application components can evolve: Web servers using

CGIBIN scripts to deliver Java applications that exchange data with RMI servers, which coordinate CICS and SQL hosts and so forth, until the web becomes too complex to manage.

What is needed is a methodical approach to evolving a simple and maintainable architecture. The method and the architecture are your choice, as a developer. In this section we look at the VisualAge for Java CICS Access Builder in the three-tier client/server context.

Presentation Logic Tier

The VisualAge for Java CICS Access Builder generates the transaction bean, which places final display and data editing in the Java user interface—which is as expected. The presentation logic tier of a three tier strategy is in the workstation.

Data Logic Tier

The host CICS enterprise system provides coordinated and controlled access to enterprise data. For some applications, such as the simple client/server ATM example, the CICS host can manage all of the enterprise data. Our data logic tier resides at the CICS host. This is fine when our Java development is a new user interface to an existing CICS system.

We might use Java to enable a new application that takes data from more than one source. For example, the ATM application might get customer details directly from an SQL database on a separate server, while using the CICS host for all account data. Future development may therefore increase the components in the data logic tier.

Business Logic Tier

In the case where we build a new presentation logic tier for an existing CICS application, the business logic will already reside in that CICS application.

Business Logic in the Workstation

When developing a new application where the data logic tier has more than one server, we might have the business logic in the user's workstation. For simple situations this will be manageable. Our biggest problem will probably be network management, because each workstation will be using communication paths to two or more servers.

This could become very complex if there are several servers; for example, our ATM application might get its account information from two central CICS servers, one for checking accounts, and one for savings accounts, while getting the customer information from one of several SQL databases, depending on the country from which the customer requests the ATM transaction. This might be workable, but it is potentially unmanageable. The workstation business will need to take the network complexity into account in its logical unit of work (LUW) commit strategy, as well as handling communications failures.

Business Logic in the Server

Where the business logic resides in the data servers, our workstation presentation logic may be simplified to only marshaling and demarshaling the data to be exchanged with each server. The presentation logic is simpler, but the business logic is now distributed across several data servers. Such a configuration requires a careful approach to application change-control, because the servers and the workstation must keep their application components synchronized.

Business Logic in an RMI Server

We can put an RMI server in between the workstation presentation logic tier and the host servers' data logic tier. If we put the business logic tier here, we have a manageable network topology. Each workstation connects to one RMI server. Each RMI server connects to the necessary data servers, and we have a fixed arrangement of connections to monitor to ensure that the RMI servers are working.

Existing host-based systems, with the business logic tier at the host server, can evolve to this topology by first introducing an RMI server to marshal and demarshal data from the host servers and then evolving the RMI server's function to include progressively more business logic.

As such an RMI server tier evolves, its usage will grow, and so the resources it needs will grow. One of the outstanding benefits of using a Java RMI is that such a server can grow from perhaps a humble Windows workstation through UNIX servers up to a complete enterprise system—all using the same write-once-run-everywhere Java code.

9

C++ Servers and C++ Access Builder

Although Java is a great language, there are situations where Java alone cannot meet all needs or where programmers are faced with a legacy environment that they want to integrate into new applications.

Sun has defined an API that allows Java code to interoperate with applications and libraries written in other programming languages, such as C or C++. However, a programmer's task to achieve this integration is not easy. VisualAge for Java contains the C++ Access Builder for generating Java beans from C++ code, so that programmers do not have to deal with this API.

In this chapter we briefly explain the concepts of the Java Native Interface (JNI) and develop sample applications to illustrate the use of the C++ Access Builder. We can then apply this knowledge to our ATM application by wrapping existing C++ server code.

Java Native Interface Overview

The objective of the Sun team, as stated in the JNI specifications, was to offer an interface to interoperate with applications and libraries written in other programming languages:

- ❑ In a standardized way across all Java Virtual Machine (JVM) implementations
- ❑ To make the time critical capabilities of languages such as C or C++ available to Java programs

The previous level of JNI in JDK 1.0 has been changed and improved with JDK 1.1. We deal here with the JDK 1.1 level.

According to the specifications, the JVM can invoke native methods following a standard naming and calling convention, and native functions can access Java objects through standard interface functions.

When to Use?

Here are some examples of when to use the JNI:

- ❑ Access to platform dependent feature
- ❑ Reuse of a legacy server written in C++
- ❑ A device already has a driver that needs to be accessed from Java.
- ❑ Some portion of the application is CPU intensive and therefore needs to be compiled for faster execution.

Before you consider using JNI to develop new applications, you must carefully evaluate the following issues:

- ❑ You lose the benefit of the Java write once, run everywhere paradigm. The code is tightly linked to some operating-system-dependent libraries.
- ❑ You give up the Java intrinsic security features. The code can access memory outside the sandbox of the JVM, you can manipulate pointers, and arrays and strings do not check for bounds. Therefore the Java class cannot be downloaded as an applet.
- ❑ The development teams must master two different environments that are close enough to each other to create confusion.
- ❑ The application will be difficult to maintain.

- ❑ A careful design using efficient class libraries, which leads to leaner code, can often overcome the performance penalty of a Java-interpreted environment.
- ❑ Just-in-time compilers become faster with every new release of the JDK, and soon static compilers will be commercially available.

After all that, you may wonder why you should use JNI and the C++ Access Builder. Well, everything is not perfect in the Java world, and C++ has the advantage of maturity. In addition, Java will not make years of development obsolete, and you cannot just throw away all that work and start from scratch. Like the other enterprise tools, the C++ Access Builder makes it easier for you to extend existing applications rather than rewrite them.

Java Native Interface Programming

In this section we cover the basics of programming the JNI, with examples to illustrate the main points. We describe how to access C code from Java, and Java code from C or C++.

Declaring and Loading Native Methods

A method written in a native programming language must be declared as native in its definition, for instance:

```
public native double nativeHypotenuse(int i, int j);
```

The method definition provides only the method signature, but no implementation; it is terminated by a semicolon. The implementation for this method, which returns the value of the hypotenuse of a triangle, given the length of its sides, is provided in a separate native C language source file.

We compile the C code and create a shared library or DLL (we present the necessary steps in “Simple JNI Example” on page 258). This DLL is loaded into the Java class with the following statement:

```
System.loadLibrary("NativeTest");
```

The `loadLibrary` method must be put in a Java static initializer, with the shared library DLL name as argument; here the DLL is called `NativeTest`. The name is converted to a name compatible with the platform on which you are running (Windows, UNIX, OS/2). The JVM runs this static initializer when it loads the class.

Simple JNI Example

Figure 159 shows the complete code of a simple Java application. The Java class gets its input from the command line, calls the native method, and returns the result to the standard output.

```
class NativeTest
{
    public static void main(String[] args)
    {
        NativeTest aTest = new NativeTest();
        int i0 = Integer.valueOf(args[0]).intValue();
        int i1 = Integer.valueOf(args[1]).intValue();
        double d = aTest.nativeHypotenuse(i0,i1);

        System.out.println("The value for the Hypotenuse for a triangle
of sides " + args[0] + " and " + args[1] + " is " + d);
    }

    public native double nativeHypotenuse(int i, int j);

    static {
        System.loadLibrary("NativeTest");
    }
}
```

Figure 159. Simple JNI Example: Java Code

We can write and save this code inside the VisualAge for Java IDE (or we use an editor and the javac compiler).

Now we have to use the *javah* utility program, provided with the JDK:

```
javah -jni NativeTest
```

The *-jni* option instructs *javah* to generate a JNI-compatible header file. This header file defines the function prototype that you implement, that is, the native Hypotenuse method declared in the Java class.

By default the name of the generated header file is the Java class name with a *.h* appended at the end of it. The file is located in the current directory.

Figure 160 shows the NativeTest.h header file.

```

/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class NativeTest */
#ifndef _Included_NativeTest
#define _Included_NativeTest
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:      NativeTest
 * Method:     nativeHypotenuse
 * Signature:  (II)D
 */
JNIEXPORT jdouble JNICALL Java_NativeTest_nativeHypotenuse
(JNIEnv *, jobject, jint, jint);
#ifdef __cplusplus
}
#endif
#endif

```

Figure 160. Simple JNI Example: Generated C Header File

Java_NativeTest_nativeHypotenuse is the name generated for the C function to implement.

Notice that the function has four parameters, instead of two in the Java method definition. The first parameter is always a pointer of type *JNIEnv*. It points to an array of pointers to JNI functions for accessing Java objects. It makes the JNI functions available to the native program (a more detailed explanation of this mechanism is given in the JNI specifications). The second parameter, *jobject*, is a reference to the java object (or Java class in case of a static native method). The third and fourth parameters are of type *jint* and not *int*. We cover this subject in “Type Mapping between Java and C/C++” on page 262.

JNIEXPORT and *JNICALL* are used for handling functions exported from DLLs on platforms such as Windows. We have to use the same keywords in the implementation of the program.

Now, we can implement the native method in a file named *NativeHypotenuse.c* (Figure 161).

```

#include "NativeTest.h"
/*header file given by javah -jni */
#include <math.h>

JNIEXPORT jdouble JNICALL Java_NativeTest_nativeHypotenuse
    (JNIEnv *env, jobject obj, jint i, jint j)
{
    return sqrt(i*i + j*j) ;
}

```

Figure 161. Simple JNI Example: C Function

We compile and link this file into a DLL, whose name is the name in the `System.loadLibrary` method, that is, `NativeTest.dll`. We use a makefile generated for a Windows system with the VisualAge C++ makefile generator (Figure 162).

```

# NativeTest.mak
# Created by IBM WorkFrame/2 MakeMake at 11:02:37 on 12/16/97
# The actions included in this make file are:
# Compile
# Make exp a
.SUFFIXES:

.all: \
    .\NativeTest.dll

.\NativeTest.obj: \
    C:\VAJResid\tests\NativeTest.C \
    {C:\VAJResid\tests;$(INCLUDE);}NativeTest.h
    @echo " Compile "
    icc.exe /Gm /Ti /Ge- /Gf- /Fo".\%.obj"
    /C C:\VAJResid\tests\NativeTest.C

.\NativeTest.exp: \
    .\NativeTest.obj
    @echo " Make exp and lib files "
    ilib.exe /Gi:NativeTest .\NativeTest.obj

.\NativeTest.dll: \
    .\NativeTest.obj \
    .\NativeTest.exp
    @echo " Link "
    icc.exe @<<
    /B" /de /pmttype:vio /noe /code:RX /data:RW /dll"
    /B" /def"
    /B" /nod:NativeTest.lib"
    /FeNativeTest.dll
    .\NativeTest.obj
    .\NativeTest.exp

```

Figure 162. Simple JNI Example: Makefile

Before testing the Java application, we have to ensure that the DLL is in a path accessible for loading. Then we run the Java application:

```
java NativeTest 2 3
```

The value for the Hypotenuse for a triangle of sides 2 and 3 is 3.6055

JNI Development Process

Figure 163 summarizes the JNI development process.

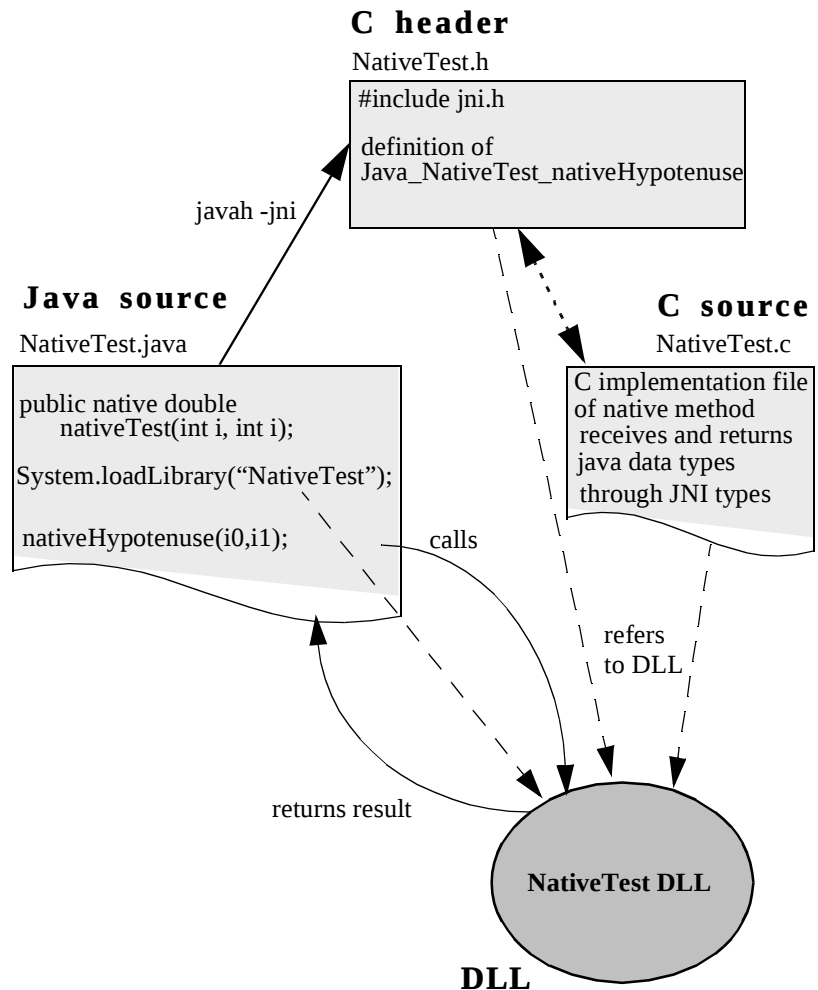


Figure 163. JNI Development Process

Type Mapping between Java and C/C++

Whenever you pass an argument in a native method or return a result, the method must be able to access or create primitive data types as well as strings, arrays, or any Java object. Therefore, JNI defines mapping between Java and C/C++ types.

Java Primitive Data Types

Native methods that use Java primitive data types, such as `int`, `boolean`, `float`, `double` as arguments, or return a value, access native types, such as `jint`, `jboolean`, `jfloat`, and `jdouble`. In our previous example, `Java_NativeTest_NativeHypotenuse` takes two `jints` as arguments and returns a `jdouble`.

Java String Types

Java strings map to `jstring`, with JNI functions to support the conversion between Java character encoding and the C `char` type. For instance, if you want to use a `jstring` named `aName` that was coded in Java Unicode, you have to write C code, such as:

```
char *aCName = (*env)->GetStringChars(env, aName,0);
print ("%s", aCName);
(*env)->ReleaseStringChars(env, aName, aCName);
```

The last statement releases the resources associated with the string.

Java Array Types

Java arrays map to corresponding types of `jarrays`. For instance, an array of integers maps to a `jintArray`. JNI functions enable you to access each element of the array:

```
jint *aCIntArray = (env*)->GetIntArrayElements(env, anIntArray, 0)
```

Remember to release the resources by using the `ReleaseIntArrayElements` function.

Accessing Java Methods and Fields from Native Code

The native method can make a call back to a Java method, using a pointer of type `JNIEnv` with JNI specific functions.

For this purpose, we must first access a Java class object. Having a pointer, `obj`, to a jobject, we must obtain the class of the object with:

```
jclass jc = (*this)->GetObjectClass(this, obj)
```

Now we can get a pointer to the method of the `jc` class:

```
anId = (env*)->getMethodId(env,jc,"ajMethod", "(II)D")
```

The last argument represents the signature of the method. `(II)D` indicates that this method takes two integers as input and returns a double. You can form the method symbolic signature according to the JNI specifications by using the `javap` tool on the Java class. This method ID can then be used for any call to the Java method.

In the same way you can get and set Java fields, with specific JNI functions. A field identifier can be retrieved with a `GetFieldID` function, giving the field class, name, and type signature. The field type signature, defined by Java type, can be found in the JNI specification or by using the `javap` tool.

Exception Handling

Java handles exceptions in a very clean way with try and catch blocks, but this is not the case for most native programs. Therefore the JNI provides functions to check for possible exceptions after a JNI function returns. JNI also provides functions to throw Java exceptions that can be handled by the JVM.

The JNI specification insists that it is important to handle exceptions from a native call, before a new native call is issued; otherwise you may get unexpected results.

Object References and Java Garbage Collector

Another major difference between Java and the native programming languages is that Java has a garbage collector that cleans up unreferenced objects. In our previous examples, we created data types such as `jstring` or `jobject` to reference Java objects. This would prevent the Java objects from being garbage collected unless the native code frees them.

Fortunately, objects passed to or returned from native methods are local references. Local references end when the native function returns. The Java garbage collector can then free the resources. But this means that consecutive native methods cannot reuse a local reference. For that purpose, native methods can create a global reference, but they have to explicitly free the references when they are no longer needed. JNI provides functions such as `NewGlobalRef` or `DeleteGlobalRefs` to ensure that these cleanup tasks are accomplished.

How to Make Your Life Easier?

We hope that by now you understand the basics of native programming in Java, that is:

- ☐ You are aware of the limitations imposed by security and portability.
- ☐ You know the restrictions concerning the types and functions you can access.
- ☐ You can evaluate the tasks to make use of native calls.

It is now time to explain how the C++ Access Builder of VisualAge for Java can help you write applications that use the JNI, and save you work by providing you with clean wrapping of C++ code.

C++ Access Builder Overview

JNI alone does not address the instantiation of C++ classes, access to methods and data members of C++ objects from Java, passing of objects by value or reference, and many other operations of C++ development.

The C++ Access Builder is a tool designed to simplify the task of calling existing C++ code from Java.

High-level View

The C++ Access Builder generates three files per C++ class:

- A C++ wrapper that maps method invocations from a Java representative class to the matching method of the corresponding C++ class.
- A Java bean that represents the public interface of a C++ class. It can be used from other Java beans to access the C++ object. The Java bean contains native method signatures for each public method in the corresponding C++ class, and set and get methods for public data members. The native methods in the Java bean are implemented in the matching C++ wrapper file.
- A makefile to compile and link the C++ wrapper into a DLL and to compile the Java bean source into a class file.

The C++ Access Builder is implemented as a command line utility, `ivj2cpp`, that generates the Java bean, the C++ wrapper, and the makefile.

The generated Java bean can be imported into VisualAge for Java and used in the Visual Composition Editor through standard connections. The public methods and data members of the original C++ object are visible as the interface of the Java bean.

The generated source code works in conjunction with the JNI Specifications, Release 1.1.

A run-time class library provides support to the access beans and associated files generated by the C++ Access Builder. This class library, called `ivjdjs10.dll`, must be deployed with the application when running outside of the VisualAge for Java development environment.

As for any application using the Enterprise Access Builder Java classes, the `ivjeab.zip` file must be in `CLASSPATH` so that classes from `COM.ibm.ivj.eab.j2cpp` are accessible.

Figure 164 shows the code generation process of the C++ Access Builder.

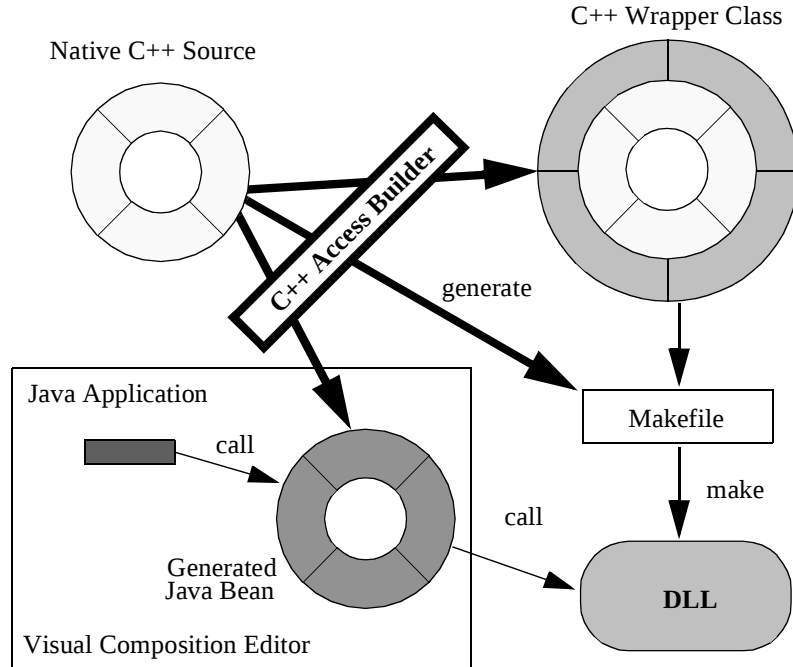


Figure 164. C++ Access Builder Code Generation

Command Line Utility

The `ivj2cpp` command line utility takes C++ source and header files as input and generates a Java bean in a Java package and a DLL as output:

```
ivj2cpp outputname headerfile.hpp -s source.cpp -p packagename
```

The input files are the header file (*headerfile.hpp*) and source code (*source.cpp*). The final output of the process is the DLL (*outputname.dll*) and the Java bean (*outputname.java*) in the Java package (*packagename*). The DLL is produced by running the makefile that is generated.

For a complete listing of the possible flags see the online documentation of VisualAge for Java, or type `ivj2cpp -h`.

We can now test this tool with the simple example that we developed in “Declaring and Loading Native Methods” on page 257.

Revisiting the Native Example with the C++ Access Builder

Let us apply the C++ Access Builder to our very simple nativeHypotenuse example.

First we need to make the hypotenuse function a method of a C++ class, which should be the case anyway in any real-life application where a company wants to reuse an existing set of complex mathematical calculation classes that have been developed over time.

The nativeHypotenuse method is embedded in a nativeops.cpp class:

```
#include "nativeops.hpp"
#include <math.h>
class _Export nativeops;
double nativeops::nativeHypotenuse(int i, int j)
{
    return sqrt(i*i + j*j);
}
```

The nativeops class has a corresponding nativeops.hpp header file:

```
#pragma library("nativeops.lib")
class nativeops
{
public:
    double nativeHypotenuse(int i, int j);
};
```

From the directory where the nativeops.hpp and nativeops.cpp files are located, we invoke the C++ Access Builder on the command line:

```
ivj2cpp nativeops nativeops.hpp -s nativeops.cpp -p opsserver
```

The first parameter is the name of the DLL to create. The p flag indicates that the generated Java bean should be in the opsserver package.

The following files are generated:

- ❑ nativeops.java, the Java bean that represents the public interface of the native class, for use in the client applet
- ❑ nativeopsWrapper.cpp, a C++ wrapper called by the bean
- ❑ nativeops.mk, a makefile to compile and link the C++ code (making a nativeops.dll) and to compile the java source into nativeops.class
- ❑ Makefile, a makefile to launch nativeops.mk

The generated files get their names from the C++ classes declared in the header file.

We compile the C++ object and generate a new DLL by entering on the command line:

```
nmake
```

During the compilation of the Java source, some classes of the COM.ibm.ivj.eab.j2cpp package are used and must be accessible with the CLASSPATH variable. They are located in:

```
\IBMJava\Ide\program\lib
```

To access the generated DLL, a definition file (.def) and a shared library file (.lib) are also generated. Moreover, on Windows 95 and Windows NT, an export file (.exp) is generated.

Now we are ready to test the native code. Note that we have not written a single line of JNI functions; everything has been done under the cover by the C++ Access Builder. Let us first test this application, and then we can look at the generated code.

We have to create a simple Java class to call the generated native-ops bean. The ivj2cpp command used the opsserver package name. We create an opsserver subdirectory and move the nativeops.class into it. We check that CLASSPATH includes the current directory (“.”). We create the OpsTest class in VisualAge for Java, or with an editor:

```
class OpsTest
{
    public static void main(String[] args)
    {
        opsserver.nativeops anOp = new opsserver.nativeops();
        int i0 = Integer.valueOf(args[0]).intValue();
        int i1 = Integer.valueOf(args[1]).intValue();
        double d = anOp.nativeHypotenuse(i0,i1);
        System.out.println("Hypotenuse of " + args[0] + " and " + args[1] +
                           " is " + d);
    }
}
```

We compile the OpsTest program and run it:

```
javac OpsTest.java
java OpsTest 3 4

Hypotenuse of 3 and 4 is 5.0
```

We discuss the details of the C++ wrapper code in “Considerations for C++ Class Wrapping” on page 272.

Reverse String Example

In the reverse string example, we create another C++ access bean by using the `ivj2cpp` utility. The purpose of this example is to show that more complex classes, such as `IString.cpp` from the IBM Open Class Library, can be used inside the C++ server objects, as long as they are not exposed in the interface.

You are now familiar with the steps to generate the Java bean, so we give only a short explanation of this process. We show you how to use this Java bean inside the Visual Composition Editor.

The `reverse.hpp` header file and the `reverse.cpp` C++ program are listed here:

```
//-----
// reverse.hpp - Header file for a C++ DLL
//-----
#pragma library("reverse.lib")
// #include <istring.hpp>
class Reverse
{
public:
    char* reverse(char* str1);
};

//-----
// reverse.cpp - Source file for a C++ DLL
//-----
#include <istring.hpp>
#include "reverse.hpp"
/*-----*/
/* This file defines 1 classe Reverse */
/* Identify the classes we wish to export */
/*-----*/
class _Export Reverse;
char* Reverse::reverse(char* str1)
{
    char* t;
    IString* myString = new IString(str1);
    t = ((*myString).reverse());
    return t;
}
```

This C++ program takes a string and reverses it by using the reverse method of the `IString` class. Instead of passing an `IString` as a parameter and return value, we pass a `char*`, because `IString` cannot be parsed by `ivj2cpp`. We explain in “Accessing a Complex Class by Header File Modification” on page 281 how a class such as `IString` can be parsed.

From the directory where the `reverse.hpp` and `reverse.cpp` files are located, we run the `ivj2cpp` utility:

```
ivj2cpp Reverse reverse.hpp -s reverse.cpp -p cppserver
```

Because we use the `IString` class in the reverse code, we need to modify the `reverse.mk` file to ensure that the VisualAge for C++ compiler uses the multithreading libraries at linkedit.

We edit the `reverse.mk` file and modify the `CXXFLAGS` line by adding the `/Gm+` flag:

```
CXXFLAGS = $(CXXINCLUDE) /Ge- /C /O /Q /Fo$* /Gm+
```

We compile the C++ server object and generate the DLL.

Now we are ready to use the C++ DLL with a Java applet. First we make sure that the directory where the DLL is located is in `PATH`, so that it can be accessed when dropping the reverse Java bean in the Visual Composition Editor; remember that the default constructor is called, and it loads the reverse DLL. If the Java constructor cannot find the DLL, the debugger opens up and we get an error message:

```
Uncaught exception  
(java.lang.UnsatisfiedLinkError : no Reverse in shared library path)
```

In the Workbench, we create a C++ Access Builder project and import the reverse Java bean code generated by `ivj2cpp`. Notice that the `cppserver` package is created automatically during import.

- ❑ We create a new applet called `ReverseApplet`, using Smart-Guide, open the Visual Composition Editor, and build the applet as shown in Figure 165.
- ❑ We add two text fields to the applet, add a push button to the bottom of the panel, and change the label to "Reverse String."
- ❑ Using the *Add Bean* menu choice, we add the reverse bean and name it `ReverseBean`.
- ❑ We connect the **Reverse String** button to the reverse method of the reverse bean (1).
- ❑ We connect the text property of the first text field bean to the `arg1` parameter of the previous connection (2). When the **Reverse String** button is clicked, the reverse method of the C++ server is called with the text of the first text field as a parameter.
- ❑ We connect the `normalResult` of the reverse call connection to the text property of the second text field (3). The resulting reversed string is sent back to the second text field.

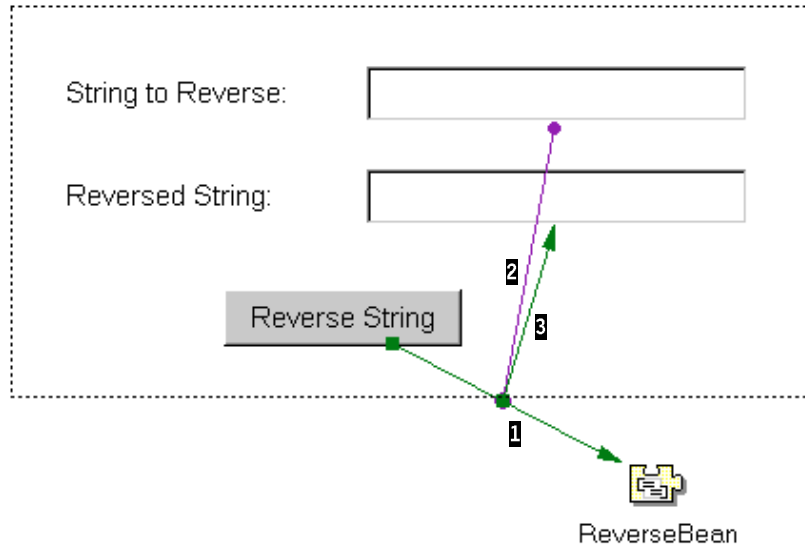


Figure 165. Test Applet for Reverse Bean

We run the `ReverseApplet` applet, type a string into the first text field, and click on the **Reverse String** button. The reversed string should be displayed in the second text field.

Note that we have been using a Java applet with JNI calls for test purposes. The applet did not break Java security rules because we are running in the same local machine. (An applet in a Web browser cannot use the JNI.)

C++ Access Builder Advanced

We have seen that the JNI defines some rules to make native C++ calls from Java. One of these rules defines a naming convention for the C++ functions. The JVM must be able to find the C++ functions, so they cannot have just any name.

Another issue is the parameter (or argument) types. The C++ and Java types are close, but they are not always identical. The C++ Access Builder creates a Java class with all the functions of a C++ class, keeping the same function names in C++ and in Java. But to do this, an intermediate C++ wrapper file has to be generated. This is the only piece of code that accesses the existing C++ objects. We explain here how the C++ wrapper works with some code extracts from the previous examples. We also explain the rules to follow to successfully generate a wrapper.

Considerations for C++ Class Wrapping

Remember that in “Java Native Interface Programming” on page 257 we used some C primitives types and function calls but did not talk about support for such things as:

- ☐ Accessing C++ objects
- ☐ Instantiation of classes
- ☐ Accessing members and methods of objects
- ☐ Overloading operators
- ☐ Passing objects by value and reference

We have seen that, given a C++ class, the C++ Access Builder generates an access bean and a C++ wrapper file to be compiled and linked as a DLL. This generation is done by parsing header files and their hierarchy of include files.

All generated Java beans inherit from a `__J2CPP_` class in the `COM.ibm.ivj.eab.j2cpp` package. This class has methods to invoke the construction and destruction of C++ and Java objects. The inheritance tree of the server class is reflected from C++ to Java as shown in Figure 166.

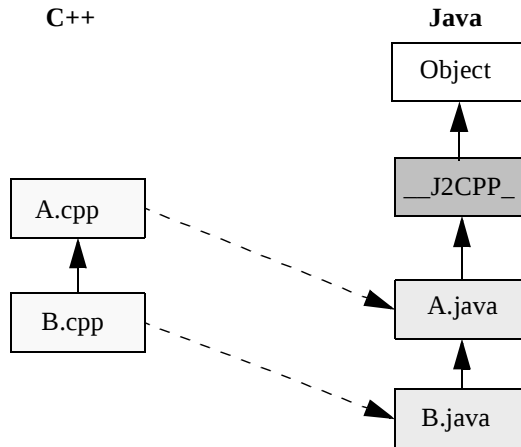


Figure 166. Mapping between C++ and Java Inheritance

All classes provided as part of the j2cpp package also inherit from `__J2CPP_`. For instance, `PCHAR`, which is a Java class representing a pointer to a char and is used when passing a `char*` as a parameter in a C++ call, inherits from `__J2CPP_`.

Details of the Generated Code

Let us go back to the Reverse string example and examine the generated code. Figure 167 shows the code for `Reverse.java`, and Figure 168 on page 275 shows the code for `ReverseWrapper.cpp`.

We do not detail all lines of the JNI code generated by the C++ Access Builder; we simply give you the flow of calls made as well as some of the classes used.

The Java class has a constructor, a reverse method, and a finalize method. The Reverse constructor calls a private native method `InitReverse_p`:

```
private native void InitReverse_p() ;
public Reverse() {
    super(_null_constructor_);
    InitReverse_p();
}
```

Here is the code to call the native method from the Java bean:

```
private native PCHAR reverse_ppC(PCHAR arg0);
public PCHAR reverse(PCHAR arg1) {
    return reverse_ppC(arg1);
}
```

The finalize method deletes the C++ object:

```
delete (Reverse*) hnd1.
```

```
/*
 * This file was generated by the IVJ2CPP tool.
 * DO NOT EDIT THIS FILE.
 */
package cppserver;
import COM.ibm.ivj.eab.j2cpp.*;
public class Reverse extends __J2CPP_ {
    protected Reverse(__NullConstructor_ _null_constructor_) {
        super(_null_constructor_);
    }
    private native PCHAR reverse_ppC(PCHAR arg0);
    public PCHAR reverse(PCHAR arg1) {
        return reverse_ppC(arg1);
    }
    private native void InitReverse_p() ;
    public Reverse() {
        super(_null_constructor_);
        InitReverse_p();
    }
    public native void finalize ();
    static {
        System.loadLibrary("Reverse");
        fixFloat();
    }
}
```

Figure 167. Code for Reverse.java

Now let us look at the generated C++ wrapper code (Figure 168). The wrapper class defines the InitReverse method as:

```
Java_cppserver_Reverse_InitReverse_1p
```

This is the name that the command `javah -jni` would generate on the `Reverse.class` file. The C++ wrapper file calls the constructor of the C++ object:

```
((jint)new Reverse())
```

PCHAR is a class provided with VisualAge for Java Enterprise and represents a pointer to a char. PCHAR can be constructed from a string and used in Java to represent the C++ `char*`. The `Reverse` method takes a PCHAR as argument and returns a PCHAR. As you will see in “Mapping the ATM C++ Classes to Java” on page 292, many classes like this are provided to map C++ types to Java classes.

```

/* This file was generated by the IVJ2CPP tool.
 * DO NOT EDIT THIS FILE.*/
#include <jni.h>
#include <stdlib.h>
#include <wchar.h>
#include <string.h>
#include "ivjddjct.hpp"
#include "ivjddjutl.hpp"
#include "reverse.hpp"
extern "C" {
JNIEXPORT void JNICALL
    Java_cppserver_Reverse_finalize(JNIEnv *env, jobject that) {
        jfieldID fid;
        jint hndl;
        jobject globobj;
        jclass cls = env->GetObjectClass(that);
        hndl = IVJDDJgetHandle(env, that);
        globobj = IVJDDJgetJavaObject(hndl);
        fid = env->GetFieldID(cls, "delete_c_object", "Z");
        if (env->GetBooleanField(that, fid) == JNI_TRUE)
            delete (Reverse *)hndl;
        IVJDDJRemoveEntry(hndl);
        if (globobj != NULL) env->DeleteGlobalRef(globobj);
    }
extern "C" {
JNIEXPORT void JNICALL
    Java_cppserver_Reverse_InitReverse_lp(JNIEnv *env, jobject that) {
        jfieldID fid;
        jint hndl;
        jclass cls = env->GetObjectClass(that);
        fid = env->GetFieldID(cls, "handle", "I");
        hndl = (jint)new Reverse();
        env->SetIntField(that, fid, (jint) hndl);
        IVJDDJEnterObjectPair(env->NewGlobalRef(that), hndl);
    }
extern "C" {
JNIEXPORT jobject JNICALL
    Java_cppserver_Reverse_reverse_1ppC
        (JNIEnv *env, jobject that, jobject arg0) {
        jclass ret_cls;
        jmethodID ret_mid;
        jfieldID ret_fid;
        jobject ret_obj;
        char *ret_val;
        char *parm0 = (char *) IVJDDJgetHandle(env, arg0);
        jint hndl;
        hndl = IVJDDJgetHandle(env, that);

        ret_val = (char *)((Reverse *) hndl)->reverse(parm0);
        ret_obj = IVJDDJGetorCreateJavaObject
            (env, "COM/ibm/ivj/eab/j2cpp/PCHAR", (jint) ret_val);
        return ret_obj;
    }
}
/*
 * End of file generated by the IVJ2CPP tool.
 */

```

Figure 168. Code for ReverseWrapper.cpp

The public reverse method calls the native method, reverse_ppC, defined in the C++ wrapper:

```
....java_cppserver_Reverse_Reverse_lppc(...) {  
.....  
    ret_val = (char *)((Reverse *) hnd1)->reverse(parm0);  
    ret_obj = IVJDJGeterCreateJavaObject  
        (env, "COM/ibm/ivj/eab/j2cpp/PCHAR", (jint) ret_val);  
    return ret_obj;  
}
```

The return value of type PCHAR can be used in the client applet as:

```
setText(String.valueOf(COM.ibm.ivj.eab.j2cpp.PCHAR result));
```

The generated code makes this mapping fully transparent.

Design Considerations

Although the C++ Access Builder improves the mapping from C++ to Java, the differences between the two languages present some limitations (see “Limitations of the C++ to Java Mapping” on page 280). You should consider creating a thin interface layer that accesses your C++ code.

If the C++ class uses multiple inheritance, a complete hierarchy mapping cannot be done because Java does not support multiple inheritance. Instead you can create a wrapper class that owns an instance of the object that is multiply inherited. The wrapper class can be defined to expose all inherited public methods. In “Another Way of Exposing the C++ Interfaces” on page 281, we explain how you can hide this inheritance from the parser and nevertheless use the C++ class and the inherited methods.

Java can only use native code provided as a DLL. When building a DLL from the wrapper files, we have two choices:

- ❑ We can rebuild the existing C++ library with the additional files. This is what we do in “Reverse String Example” on page 269. Because we have the source file of everything, we enter:

```
ivj2cpp Reverse reverse.hpp -s reverse.cpp -p cppserver
```

- ❑ We can build a new C++ library that can access the existing DLL. This is what we do in “Using a Class That Accesses a Wrapped C++ Library” on page 284 because we do not have the source code of everything (note the usage of the -l flag):

```
ivj2cpp WordReverse WordReverse.hpp -s WordReverse.cpp  
-l String.lib -p StringTest
```


Another important feature is the way in which the `ivj2cpp` utility parses the source files. Only header files are parsed, but all included header files are also processed. Therefore, the header files should not have any `#include` statement other than the statements that correspond to types passed as parameters or return values. We can put the include statements for the files that do not have to be parsed in the implementation file, just before the `#include` statement of the parsed header file.

Type Mapping between C++ and Java

Table 20 summarizes of the C++ types that are mapped to Java classes. It provides a first insight into the design decisions you have to make to create effective interfaces to C++ classes that can be processed to generate Java beans.

Table 20. C++ Primitive Type to Java Class Mapping

C++ Type	Java Class Name
char	PCHAR
unsigned char	PUCHAR
signed char	PSCHAR
wchar_t	PWCHAR
unsigned int	PUINT
signed int	PINT
unsigned short	PUSHORT
signed short	
unsigned long	PULONG
signed long	PLONG
double	PDOUBLE
float	PFLOAT

Public member variables of primitive types are made accessible with a `get` and `set` method in the Java bean (except that no `set` method is generated for the `const` keyword).

Pointers to C++ primitive types are provided as Java classes, with methods to `get` and `set` the value of the equivalent C++ value. There are 12 classes of this type. The complete list is shown in Table 20; for more details, see the reference section of the online documentation.

For arrays, get and set methods are generated to allow access to each element. For instance, if we have in C++ a matrix declared as

```
int Matrix[2][3]
```

we can set each element with

```
setMatrix(1,1,45);
```

We provide an array example in “Using a C++ Server in the ATM Application” on page 290.

The generated get and set methods for pointer data members refer to and can change the address of the corresponding C++ data member. There are also methods on each P<TYPE>.class for changing the value pointed to. Here is an example of using the PCHAR class, where the C++ class A has a constructor, A::A(char* name). The code of the Java client is:

```
String str = "C++ Access Builder";
PCHAR toolName = new PCHAR(str);
A anA = new A(toolName);
String itsName = toolName.toString();
```

References are treated the same way as pointers, with specific classes. The generated C++ wrapper classes make the difference between passing by value or by reference.

In the case of a pointer or reference to a class other than the base classes (that is, a class that has already been mapped), the C++ Access Builder generates a PointerToCLASS_<Name> or ReferenceToCLASS_<Name>. For instance, if the C++ class A has been parsed, and in the C++ class B a method, m1, returns a pointer to class A, we can write in Java:

```
B aB = new B();
PointerToCLASS_A pA = B.m1()
```

Public methods in the Java bean call corresponding native methods in the C++ wrapper and forward the calls to the methods of the C++ objects, as we have seen in “Reverse String Example” on page 269.

Additional Java files are generated if the class has static functions:

- ❑ Statics.java—contains Java access for each C++ declaration that is not part of any C++ class. (For static methods no object has to be instantiated.)
- ❑ StaticsWrapper.cpp—contains the C++ wrapper corresponding to the Java class static functions

The example described in “Reverse String Example” on page 269 also shows that a constructor and a finalize method are generated to manage the life of their corresponding C++ object. Any time a C++ object is allocated as a result of using the new operator on a wrapping Java object, the C++ object is deleted when the Java object is garbage collected. If the C++ object is created on the C++ side and returned to Java, the C++ object is (by default) not deleted when its matching Java object is garbage collected. We can, however, override this default action by calling the `EnableCDestructor` method of `__J2CPP_` in the Java code.

Note: There are cases when the C++ object holds a reference on the Java object, preventing it from being garbage collected, and therefore freeing the memory allocated by C++. The update release of VisualAge for Java (see “C++ Access Builder” on page 379) has a native delete method added to the parent class of all generated Java classes. To ensure that the resource allocated to the C++ object are freed, we can call the delete method on the Java object.

C++ overloaded operators are mapped to Java methods with special names, for example:

```
aClass operator+ (aClass A) in C++ gives x = y.Operator_PLUS(z)
```

The Java client cannot construct a union or enum or access their value. It can only receive a union or enum as a return value or parameter (by their address of type `int`). In other words, enum and union are opaque and can only be passed between Java and C++, not accessed by Java.

Cast operators are mapped to the method of the corresponding Java class, with a name that is derived from the target type of the conversion function, for example:

```
operator long() {return (long v); //C++
long b = a.operatorCAST_LONG(); //Java
```

We show an example of casting in “Accessing a Complex Class by Header File Modification” on page 281.

Exception Handling

Because `IVJJException` exceptions may be raised by classes of `COM.ibm.ivj.eab.j2cpp`, the client code must be ready to handle them.

The C++ code can also throw an exception, and the exception should be handled by the C++ wrapper code. This is not currently done in the current release but should be part of a future enhance-

ment. In “Using a C++ Server in the ATM Application” on page 290, we show an example of how to modify the generated code to handle an exception thrown by the C++ code.

Compiler Support

When generating the makefile, `ivj2cpp` assumes that the IBM VisualAge for C++ product is used for compilation and linkage. If you are using the Microsoft Visual C++ compiler, you must specify the `-v` option with `ivj2cpp`. If you are using another compiler, you must manually edit the makefile.

The online documentation of VisualAge for Java gives the conditions under which another compiler can be used.

Limitations of the C++ to Java Mapping

In regard to direct parsing of the C++ server header files, a complete and up to date list of the limitations of the C++ Access Builder is given in the online documentation of VisualAge for Java. These limitations are due in part to the inherent differences between C++ and Java

When the C++ Access Builder parses the include files it tries to map all types and functions to something meaningful in Java. We have seen that a good way to circumvent these restrictions is to write a C++ interface class, and we extend this in “Accessing a Complex Class by Header File Modification” on page 281.

Here are the features that are not supported by the C++ Access Builder:

- ☐ Object serialization—If you require object serialization, you need to wrap the generated Java classes and provide an interface that supports serialization.
- ☐ Multiple inheritance
- ☐ Templates
- ☐ Default arguments in method calls—You must supply the default values because the call is made on the Java side.
- ☐ Parsing of files that contain incorrect syntax—You may get unpredictable results if your source files contain syntax errors.
- ☐ Exceptions—C++ exceptions are not transmitted to the Java code.
- ☐ Nested class definitions or typedefs—More specifically, there is no support for class definitions or typedefs within a class definition.

- ❑ Inheritance across the Java/C++ boundary—You cannot invoke protected methods, and there is no programming model for abstract base classes.
- ❑ Direct access to data members—You must use accessory methods.
- ❑ Multidimensional arrays—Three-dimensional arrays or larger are not supported when passed as parameters.

Another Way of Exposing the C++ Interfaces

There is a way of using the C++ Access Builder other than the way described in the online help delivered with VisualAge for Java. Remember that the goal is to generate a wrapper class that can be processed by the C++ Access Builder. The solution described in the online help implies that you rewrite a new class that exposes the interface you want to make available and that can be correctly processed by the tool.

Another solution, explained here, is a good alternative. It requires some changes in the header files of the classes you want to use so that they can be processed by the C++ Access Builder. The modified header file must contain only simple C++ types or other classes that can be processed by the C++ Access Builder. Even if we introduce changes to the class definition, all the calls we invoke on the class must be unchanged.

Accessing a Complex Class by Header File Modification

As a first example, we take a C++ string class and try to make it usable from Java. For instance, working with VisualAge for C++, we often use the IString class. The definition file is in D:\IBM-CPPW\INCLUDE. We copy ISTRING.HPP to a working directory as STRING.HPP because we are going to modify it and do not want to change the original header file.

From the very long definition of Istring, we keep only some simple methods, for example:

- ❑ Three constructors (the default, the constructor from a set of characters, and the constructor from a preexisting IString) and the destructor
- ❑ The reverse method
- ❑ The cast operator to char*
- ❑ The word method that extracts one word from an IString sentence

Figure 169 shows the reduced header file.

```
class IString {
public:
//----- Constructors -----//
    IString      ( );
    IString      ( const IString &aString );
    IString      ( const char *pChar );
    ~IString     ( );

//----- Methods -----//
    IString &reverse ( );
    operator char* ( ) const;
    IString word    ( unsigned int wordNumber ) const;
};
```

Figure 169. Reduced IString Header File

We parse the modified header file with the `ivj2cpp` command:

```
ivj2cpp String String.hpp -p stringtest
```

Because we named the header file `String.hpp`, so that there is no confusion with the real `IString.hpp`, `ivj2cpp` generates `IString-Wrapper.cpp` with:

```
#include "String.hpp"
```

However, the real code is required at link time, otherwise we get many error messages. Therefore, we change the include statement to:

```
#include <IString.hpp>
```

Because the `IString` class is a multithreaded class, we have to tell the compiler to use the multithreaded library. By default the generated makefile does not set this flag. We edit `String.mk` and change the line:

```
CXXFLAGS = $(CXXINCLUDE) /Ge- /C /O /Q /Fo$*      (old)
CXXFLAGS = $(CXXINCLUDE) /Ge- /C /O /Q /Fo$* /Gm+  (new)
```

We compile the code with `nmake` and get:

- ☐ An `IString.class` Java bean in package `stringtest`
- ☐ `String.dll` and `String.lib`, which we put into a directory accessible for this type of file

We test the bean inside the Visual Composition Editor of VisualAge for Java. We create an applet called `StringUsage` and add the buttons, text fields, and labels as shown in Figure 170.

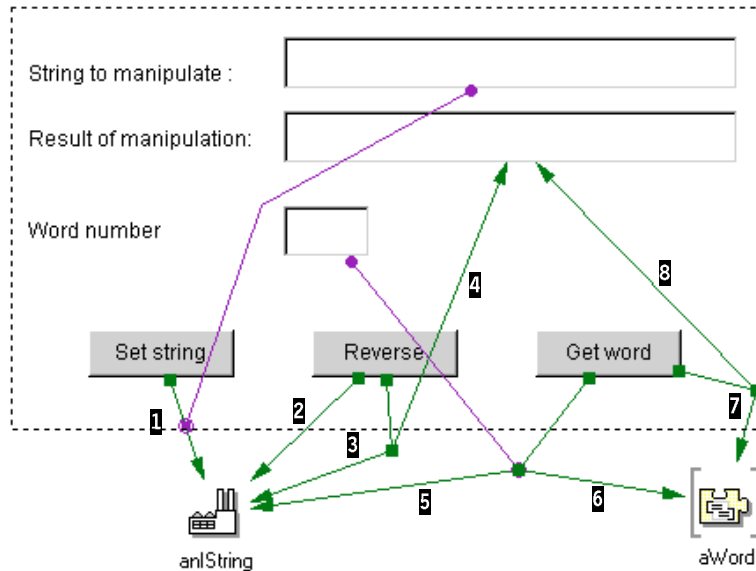


Figure 170. Applet Using the IString Bean

We drop two nonvisual components:

- ❑ A factory. We change its type to `StringTest.IString` and name it `anIString`. This component is used to instantiate an `IString` with the value of an entry field.
- ❑ A variable. We change its type to `StringTest.IString` and name it `aWord`. This component holds a reference to the `IString` that is created by the word method of `IString`.

Now we can draw the connections:

- ❑ **Set string** button to `IString(PCHAR)` constructor of `anIString` (1), passing the text of the first entry field as a parameter (`PCHAR` and `String` can be safely connected)
- ❑ **Reverse** button to reverse method of `anIString` (2)
- ❑ **Reverse** button to `operatorCAST_ConstPointerToCHAR` method of `anIString` (3), and the normalResult of this connection to text of the second entry field (4)
- ❑ **Get word** button to word method of `anIString` (5), passing the text of the third entry field as a parameter, and the normalResult of this connection to this of `aWord` (6)
- ❑ **Get word** button to `IString(PCHAR)` of `aWord` (7), and the normal result of the connection to the text of the second entry field (3)

We save and run this applet. We enter a string with blank separators between each word, click on the **Set string** button, then the **Reverse** button, and verify that the applet works correctly. We enter a number, click on **Get word**, and we should get the corresponding word in the sentence.

We have demonstrated that it is possible to access a complex C++ class with the C++ Access Builder by modifying the header file of the class.

Using a Class That Accesses a Wrapped C++ Library

Let us now reuse a piece of C++ code that has an `IString` as a parameter or a return value. We describe how we can make it work. This sample program takes an `IString` as input and returns one word of this `IString` reversed (not a very useful example, but it uses the methods we have mapped and shows all the principles).

Figure 171 shows the header file (`WordReverse.hpp`) and the source file (`WordReverse.cpp`).

Notice that the header file includes `String.hpp` and not `IString.hpp`, so that the parser accesses the modified file. We use the `ivj2cpp` utility to generate a new C++ library named `WordReverse` that can access the existing C++ library `String`:

```
ivj2cpp WordReverse WordReverse.hpp -s WordReverse.cpp -l String.lib  
-p StringTest
```

The generated makefile, `WordReverse.mk`, rebuilds the `IString` objects. However we already did that with modifications in the wrapper code, so we remove all references to making these objects (in `GENCFILES`, `GENJFILES`, `OBJECTS`, `INTOBJECTS`) with an editor and add `/Gm+` to `CXXFLAGS` as in the previous example.

The linker needs to access the real `IString`, therefore we change `WordReverse.hpp` back to:

```
#include <IString.hpp>
```

We create a subdirectory of the name of the Java package (`StringTest`), put the `String.class` in it, and make sure that the `CLASSPATH` includes the current directory.

Now we run `nmake` to compile the classes and link the DLL. We make the generated DLL and LIB file available through `PATH`.


```

/*-----*/
/* WordReverse.hpp header file */
/*-----*/

#pragma library("WordReverse.lib")
#include <String.hpp>
class WordReverse
{
public:
    WordReverse();
    WordReverse(const IString & str);
    IString getWR(int i);
private:
    IString aString;
    IString aWord;
};

//-----
// WordReverse.cpp - Source file for a C++ DLL
//-----
#include "String.hpp"
#include "WordReverse.hpp"

class _Export WordReverse;

WordReverse::WordReverse() {}
WordReverse::WordReverse(const IString & str)
{
    aString = str
}
IString WordReverse::getWR(int i)
{
    aWord = aString.word(1);
    return aWord.reverse();
}

```

Figure 171. WordReverse Header and Source Files

We test the new bean with VisualAge for Java in an applet called WRTTest.

First we import the new WordReverse.class bean into the Workbench and drop three nonvisual beans on the free-form surface:

- ❑ A factory of type IString, named anIString
- ❑ A factory of type WordReverse, named aWordReverse
- ❑ A variable of type IString, named aWord

See Figure 172 for the layout of the applet.

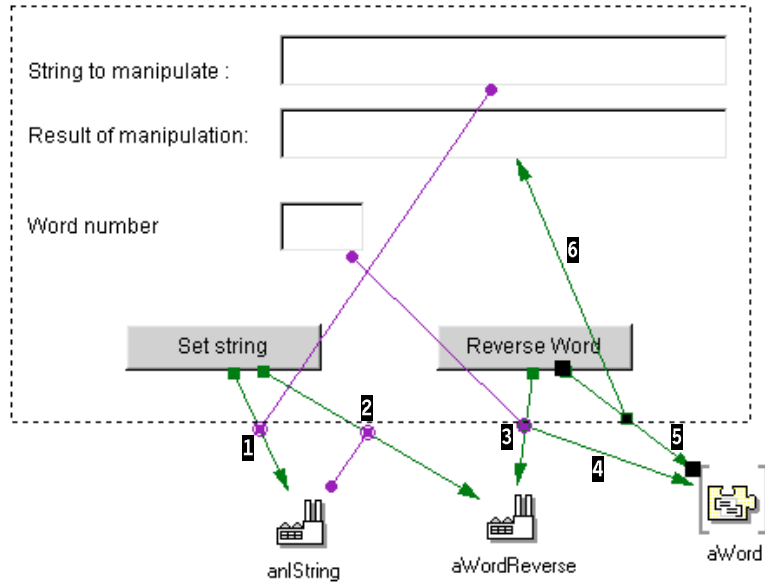


Figure 172. Applet Using the WordReverse Bean

We draw the following connections:

- ❑ **Set String** button to the IString(PCHAR) constructor of anIString (1), passing the text of the first entry field as a parameter (PCHAR and String can be safely connected)
- ❑ **Set String** button to the WordReverse(StringTest.IString) constructor of aWordReverse (2), passing this of anIString as a parameter
- ❑ **Reverse Word** button to the getWR method of aWordReverse (3), passing the text of third entry field as a parameter, and the normalResult of this connection to this of aWord (4)
- ❑ **Reverse Word** button to the operatorCAST_PointerToCHAR method of aWord (5), and the normalResult of the connection to the text of the second entry field (6)

We save and run the applet. We enter a string with blank separators between each word, click on the **Set string** button, give the number of the word we want to reverse, and click on the **Reverse Word** button.

We hope it has become clear how we can help the C++ Access Builder parse complex classes by simplifying the header files. With the technique we illustrate, we can use classes from a library as parameters of other classes and therefore reuse parts of an existing C++ project.

Accessing a Class with Templates

Let us now illustrate how we can work with template classes. Figure 173 shows the full header file of a SortedList class, which we want to use in VisualAge for Java to sort Java strings.

```
#include <IString.hpp>
#ifndef LISTH
#define LISTH
template <class T> class Element;
template <class T> class List
{ public:
    List(T& elmt);
    ~List();
    virtual void add( T& elmt );
    int getSize();
    T& goFirst();
    T& getCurrent();
    T& goNext();
    virtual void destroy();
    int size;
    Element <T> * first;
    Element <T> * current;
};
template <class T> class SortedList : public List <T>
{ public:
    SortedList( T& elmt );
    virtual void sort();
};
class SortedListString : public SortedList <IString>
{ public: SortedListString(IString& elmt);
};
template <class T> class Element
{ public:
    Element(T *elmt, Element <T> *prv, Element <T> *nxt);
    ~Element();
    Element <T> *getNext();
    Element <T> *getPrevious();
    T *getData();
    void setNext(Element <T> *nxt);
    void setPrevious(Element <T> *prv);
    protected:
    T * data;
    Element <T> * next;
    Element <T> * previous;
};
#endif
```

Figure 173. Full Header File of a SortedList C++ Class

Figure 174 shows the subset that we need to expose to the C++ Access Builder. This code uses the IString class that we built.

```
#include "String.hpp"
class SortedListString
{
public:
    SortedListString( IString& elmt );
    ~SortedListString();
    virtual void add( IString& elmt );
    virtual void sort();
    int getSize();
    IString& goFirst();
    IString& getCurrent();
    IString& goNext();
};
```

Figure 174. Reduced SortedList Header File

There is a big difference between the two header files, but, when you look carefully, you notice that only the unnecessary information has been removed.

For use in VisualAge for Java, the parser does not need to know which class is the parent class of SortedList; therefore we remove the inheritance declaration. However, we have to redeclare the parent features that are inherited and that we plan to use.

In Java, we can never use the List <T>, SortedList <T>, or Element <T> classes; therefore we hide these definitions.

In Java, we cannot use the private and protected features. Therefore we can omit these declarations. (This does not mean that Java does not support private or protected features. It simply means that all of the Java classes we create are not allowed to access these private and protected data and functions because Java classes do not inherit from this class.)

We also remove any unnecessary #define or #include statements.

C++ Access Builder Supported Environments

The C++ Access Builder generates some code that is specific to a compiler. In the C++ wrapper files, there is a `#include` statement for including wide character functions. For VisualAge for C++, this file is `wcstr.h`; for Microsoft Visual C++, this file is `wchar.h`.

If the C/C++ compiler that is used has a different header file for wide character functions, modify the `#include` statement as appropriate in each generated C++ wrapper file.

The supported IBM C++ compilers are:

- ❑ VisualAge for C++ 3.5 with Fixpak 2 on Windows 95 or NT 4.0
- ❑ VisualAge for C++ 3.0 with Fixpak 6 on OS/2 Warp Version 4

The OS/2 JDK has implemented the JNI jlong as a structure with C functions to perform casting operations. Therefore, the generated C++ wrapper code is different on OS/2 and is not portable to Windows 95 or NT 4.0.

The C++ Access Builder has been tested with and supports the following or later versions of the Microsoft Visual C++ compiler:

- ❑ Visual C++ 5.0 on Windows 95 or NT 4.0

Using a C++ Server in the ATM Application

You have already seen many ways of building and accessing server objects from a Java applet in the ATM application.

In this section we replace one of those server objects with a Java bean that accesses a C++ library, and we show that it can be done quite easily with the C++ Access Builder.

Indeed, one of the great strengths of C++ is its speed of execution, and we could, for example, develop an interest calculation program, but that would not teach you more about the tool, because we would basically have some numbers as input and a number as output.

We want to show you that it is possible to integrate an existing C++ business object in the same way as we have done with a Java program.

There is one server object in the ATM application that fits well in this context, the card object. The card object should be kept secret, because it holds the relationship between the card number, the customer ID, and the accounts of the customer and can check for the validity of the PIN.

Environment

Let us assume that there is a C++ application program, which was written a long time ago, and which the company cannot change because it must run on a specific computer with limited access.

We simulate this situation by providing a DLL and its associated header files. This program is not meant to be elaborate and perfectly written; it is just meant to illustrate that we can access a C++ DLL easily from VisualAge for Java, even without having its source code.

The C++ program stores all of the card information in a flat file and retrieves the information with the help of some methods. The C++ classes of this program are somewhat similar to what the Data Access Builder generates when we map the card table. In this C++ program, a CardManager can retrieve a row from a file, and a Card object can be constructed to hold the card, customer, and account information associated with one ATM card.

Approach

We show you how to map the CardManager and Card C++ classes with the C++ Access Builder to corresponding Java beans, and we implement a sample applet that tests the function of these beans.

For the ATM application we use some of the methods of these beans to demonstrate how a C++ server could replace the JDBC database access used in Chapter 7, “ATM Application with RMI”.

C++ Header Files

The existing C++ card class can validate a PIN and retrieve the customer and the accounts associated with the card.

The implementation consists of two classes, CardManager and Card, in a CardAccess library. The header files are shown in Figure 175 (CardManager) and Figure 176 (Card).

```
#pragma library("CardAccess.lib")

#include <istring.hpp>
#include <fstream.h>
#include "Card.hpp"

class CardManager
{
public:
    CardManager();
    ~CardManager();
    CardManager & close();
    CardManager & open(IString newFileName);
    CardManager & readLine();
    CardManager & setCurrentLine(const IString & aNewLine);
    CardManager & setFileName(const IString & aFileName);
    IString fileName() const;
    IString currentLine() const;
    IString cardLine(const IString & aCardId) ;

private:
    IString iFileName;
    IString iCurrentLine;
    fstream aFile;
    Boolean eofReached;
    Boolean CardFound;
    Boolean aFileIsOpen;
    IString endOfFile;
};
```

Figure 175. CardManager Header File

The `CardManager` class can open and close a file, read a line, return the current line, and return the line that matches a card ID. It has private methods that we do not expose to Java but that will be used in the interface code.

If the file cannot be opened, the C++ code raises an exception. If the card cannot be found, the C++ code returns “NoCardFound.”

```
#pragma library("CardAccess.lib")

#include <istring.hpp>

class Card
{
public:
    Card(ISTring aLine);
    ~Card();
    const IString cardNumber();
    const IString customerId() ;
    IString* accounts() ;
    Boolean checkPin(const IString & aPin) ;
private:
    IString iCurrentLine;
    IString iCardNumber;
    IString iCustId;
    Boolean PinOk;
    IString iAccounts[10];
};
```

Figure 176. Card Header File

The `Card` class constructs a card object from a line read by the `CardManager`. The card object can return its card number, the customer ID of its card holder, and a list of associated bank accounts. The card also has a `checkPin` method to validate the PIN.

Mapping the ATM C++ Classes to Java

Because these header files contain types that are not directly understood by the `ivj2cpp` utility, we write interface classes to access the methods and return valid data types. We do this rather than using header file reduction, because of the size of these interfaces.

We create two new classes, which we call `J_CardManager` and `J_Card`.

Mapping the CardManager Class

We create the J_CardManager class from the CardManager class:

- ❑ In the header file we omit any inclusion of files and instead put all #include statements into the source file:

```
#include <istring.hpp>
#include "J_CardManager.hpp"
#include "CardManager.hpp"
```

- ❑ We transform IString into char*:

```
IString cardLine(const IString & aCardId)      (old)

char* cardLine(char* aJCard);                  (new)
```

- ❑ We declare two private pointers to a line and a file. These pointers can only be of type char or void.
- ❑ We create a constructor and a destructor that call the constructor and destructor of the C++ class (the pointer to a CardManager class must be cast to void):

```
J_CardManager::J_CardManager() {
    CardManager* aFFile = new CardManager();
    aFF = (void *)aFFile;
}
J_CardManager::~J_CardManager() { delete aFF;}
```

- ❑ We access the methods of CardManager with this pointer, which we have to recast to its correct type. The same mechanism is true for the pointer to a line, for instance:

```
char* J_CardManager::cardLine(char* aJCard)
{
    return((char*)((CardManager*)aFF)->cardLine((IString) aJCard));
}
```

Figure 177 shows the header file and Figure 178 shows the source of the J_CardManager class. We add two additional methods for test purposes, currentLine and fileName.

```
class J_CardManager {
private :
    char* aJline;
    void* aFF;
public :
    J_CardManager();
    ~J_CardManager();
    char* currentLine();
    char* cardLine(char* aJCard);
    char* fileName();
    J_CardManager & open(char* JnewFileName);
    J_CardManager & close();
};
```

Figure 177. CardManager Interface Class Header File

```

#include <istring.hpp>
#include "J_CardManager.hpp"
#include "CardManager.hpp"

class _Export J_CardManager;

J_CardManager::J_CardManager()
{ CardManager* aFFile = new CardManager();
  aFF = (void *)aFFile; }
J_CardManager::~J_CardManager() { delete aFF; }
char* J_CardManager::currentLine()
{ return ((char*) ((CardManager *)aFF)->readLine().currentLine()); }
char* J_CardManager::cardLine(char* aJCard)
{ return ((char*) ((CardManager*)aFF)->cardLine((IString) aJCard)); }
char* J_CardManager::fileName()
{ return ((char*) ((CardManager *)aFF)->fileName()); }
J_CardManager & J_CardManager::open(char* JnewFileName)
{ ((CardManager*)aFF)->open((IString) JnewFileName);
  return *this; }
J_CardManager & J_CardManager::close()
{ ((CardManager*)aFF)->close();
  return *this; }

```

Figure 178. CardManager Interface Class Source Code

Mapping the Card Class

The process we used to map the CardManager applies here as well; we define a C++ interface class with types that the C++ Access Builder can understand.

Two additional data types must be handled: the Boolean data type and the array of IString:

- ❑ The checkPin method takes a char* as input and returns an int, 1 for true, 0 for false.
- ❑ In the current release of the product, array declaration is supported only as type x[][num], and not as type* x[num]. In our program we would like to retrieve an array of character strings, char* iAccounts[10]. We made a slight modification in the interface class, as indicated below. Although it retrieves all the accounts initially, it returns only one in each call, given the index as a parameter in the accounts method. The Java client has to call the methods as many times as there are accounts to fill up the list.

```

IString* accounts()      // method declaration in Card.hpp

char* accounts(int i)    // method declaration in J_Card.hpp

// implementation in J_Card.cpp
IString* astrAccounts = ((Card*)aC).accounts();
return (char*)astrAccounts[i];

```

Figure 179 shows the header file and Figure 180 shows the source of the J_Card class.

```
class J_Card
{
private:
    char* aJLine;
    void* aC;
    int PinOK;
public:
    J_Card(char* aCLine);
    ~J_Card();
    const char* customerId();
    const char* cardNumber();
    char* accounts(int i);
    int checkPin(char* aJPin);
};
```

Figure 179. Card Interface Class Header File

```
#include <istring.hpp>
#include "J_Card.hpp"
#include "Card.hpp"

class _Export J_Card;

J_Card::J_Card(char* aCLine)
{ Card* aCard = new Card((IString) aCLine);
  aC = (void *) aCard; }
J_Card::~J_Card() { delete aC; }
const char* J_Card::customerId()
{ return ((char*) ((Card*)aC)->customerId()); }
const char* J_Card::cardNumber()
{ return ((char*) ((Card*)aC)->cardNumber()); }
const char* J_Card::accounts(int i)
{ IString* astrAccounts = ((Card*)aC)->accounts();
  return (char*)astrAccounts[i]; }
int J_Card::checkPin(char* aJPin)
{ if (((Card*)aC)->checkPin((IString) aJPin) == true) return PinOK = 1;
  else return PinOK = 0; }
```

Figure 180. Card Interface Class Source File

We run ivj2cpp on these two interface classes:

```
ivj2cpp J_CardAccess J_Card.hpp J_CardManager.hpp
-s J_Card.cpp J_CardManager.cpp -p CardServer
```

We check that the ivj2cpp.log was generated without error messages.

If we take a look at the C++ source code we can see that the open file method can throw an exception:

```
IAccessError exc = IAccessError("Could not open file : " + iFileName);
ITHROW(exc);
```

Currently the C++ Access Builder does not generate a try and catch block in the wrapper code, although it is planned for a later release. Therefore, we must do it ourselves in the meantime.

We find that the open method in J_CardManager.java calls a private native method, called *private native open_ppC(PCHAR arg2)*. If we run `javah -jni` on the compiled class file, we find that the native method in the header file is called:

```
Java_cardserver_J_1CardManager_open_1ppC(JNIEnv *env, jobject that,
                                           jobject arg0)
```

We look for this name in J_CardManagerWrapper.cpp:

```
J_CardManager& ret_hndl = ((J_CardManager *) hndl)->open(parm0);
```

We surround this call with a try and catch block:

```
try {
    J_CardManager& ret_hndl = ((J_CardManager *) hndl)->open(parm0);
    ret_obj = IVJJDJGetorCreateJavaObject (env,
                                           "cardserver/J_CardManager", (jint) &ret_hndl);
}
catch(...) {
    jclass ExceptionClass = env->FindClass
        ("COM/ibm/ivj/eab/j2cpp/IVJJException");
    env->ThrowNew(ExceptionClass,
                  "C++ call to open threw an exception");
    return NULL;
}
return ret_obj;
```

The catch clause catches any exception thrown by the call to open. The two lines within the catch are JNI calls. The first call finds the name of an exception class. We use the class that is provided by the C++ Access Builder class library. The second call constructs an exception object from the given class, with the message string passed, and then throws the exception. The exception can then be handled in the java code that made the call to open.

In Java code J_CardManager.java, we indicate that open_ppC and open throw an IVJJException:

```
private native J_CardManager open_ppC(PCHAR arg2) throws IVJJException;
public J_CardManager open(PCHAR arg3) throws IVJJException
```

We compile and link the application by issuing `nmake` on the command line and ensure that the CardAccess and J_CardAccess DLLs can be accessed with the PATH environment variable.

We test the two generated beans and the DLL with an applet in VisualAge for Java.

Testing the Card Beans

In the same project where we developed the previous examples, we import the generated Java beans, `J_Card.class` and `J_CardManager.class`.

We create a new applet called `CardTest`, open the Visual Composition Editor, and draw the graphical interface with buttons, text fields, and an `IList` for the list of accounts (Figure 181).

We also drop five nonvisual beans:

- ❑ A factory of type `PCHAR`, named `aFileName`
- ❑ A bean of type `J_CardManager` (`J_CardManager1`)
- ❑ An `IMessageBox`
- ❑ A variable of type `PCHAR`, which holds a reference to a card, called `TheCard`
- ❑ A factory of type `J_Card`, named `aCard`

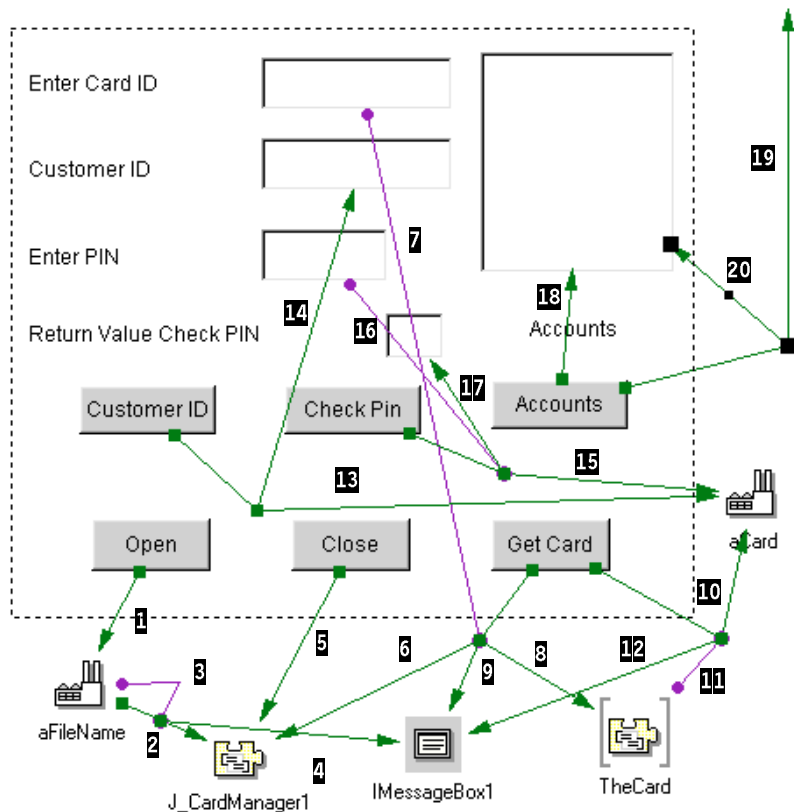


Figure 181. Test of Generated Beans for the ATM Application

We draw the connections illustrated in Figure 181:

- ❑ **Open** button to PCHAR(java.lang.String) of aFileName (1), passing the full path name of cards.txt file as a parameter (c:\VAJResid\CPPCard\cards.txt)
- ❑ The this event of aFileName to open(PCHAR) of J_CardManager (2), passing this of aFileName as a parameter (3), and exceptionOccurred of the connection to the showException method of the IMessageBox (4) (passing the event data)
- ❑ **Close** button to close method of J_CardManager (5)
- ❑ **Get Card** button to cardLine(PCHAR) method of J_CardManager (6), passing the text of the Card ID text field as a parameter (7), the normalResult of the connection to this of TheCard (8), and the exceptionOccurred of the connection to showException of the IMessageBox (9) (passing the event data)
- ❑ **Get Card** button to the J_Card(PCHAR) constructor of the aCard Factory (10), passing this of TheCard as a parameter (11), and the exceptionOccurred of the connection to showException of the IMessageBox (12) (passing the event data)
- ❑ **Customer ID** button to the customerId method of aCard (13), and return the normal result to the Customer ID text field (14)
- ❑ **Check Pin** button to the checkPin method of aCard (15), pass the text of the Enter Pin text field as a parameter (16), and the normalResult to the Return Value Check PIN text field (17)
- ❑ **Accounts** button to removeAll method of the IList (18)
- ❑ **Accounts** button to the CardTest applet as event-to-script (19) to a new accounts method to build a vector of accounts from each call to the wrapped C++ code:

```
public java.util.Vector accounts() {  
    java.util.Vector result = new java.util.Vector();  
    for (int i=0; i<10; i++) {  
        java.lang.String anAccount =  
            String.valueOf(getaCard().accounts(i));  
        if (anAccount.compareTo("0")==0) break;  
        result.addElement(anAccount);  
    }  
    return result;  
}
```

- ❑ normalResult of previous connection (19) to the elements property of the IList (20)

We save and test the applet and check that the result are correct against the cards.txt file. Here is the test sequence:

1. Click on **Open** to read the cards.txt file.

2. Enter a card ID (1111111), and click on **Get Card**.
3. Click on **Customer ID** to fill the customer ID field (101).
4. Enter a PIN (1111), and click on **Check Pin**. The result (in the Return Value Check PIN field) is either 1 (OK) or 0 (failed).
5. Click on **Accounts** to fill the list box with account numbers (101-1001, etc).
6. Click on **Close**.

Against a nonexistent cards.txt file, the applet should not break but should show an exception in the IMessageBox with this message:

```
C++ call to open threw an exception
```

Wrapping the Beans for the ATM Application

The two Java beans cannot be put directly inside the ATM application; they need to have an interface defined so that the controller of the ATM application can delegate the work to the new beans.

C++ Card Server Bean

We create a CPPCard bean that integrates all C++ functions with the IDE of VisualAge for Java. The public interface of the CPPCard bean consist of:

- ❑ CPPCard(java.lang.String)—a constructor that retrieves the card information from the cards.txt file
- ❑ cardId and customerId—two readable properties of type String, not bound
- ❑ accounts—a readable property of type Vector (account numbers), not bound
- ❑ checkPin(String)—a method returning a boolean

In addition to the public interface, there are a private field and a private method:

- ❑ aJ_Card—a field of type J_Card (the Java bean generated for the Card class)
- ❑ fillTheAccounts—a method to fill the accounts vector

Most of the code is generated with SmartGuide by creating the properties. The tailored code can be cut and paste from the CardTest applet with minor modifications.

The most important method is the constructor. The constructor allocates the `J_CardManager` and calls it to open the file and retrieve the card with the given card ID. Then it allocates the `J_Card` and calls it to retrieve the customer ID and list of accounts:

```
public CPPCard (java.lang.String aCardId) {
    try {
        COM.ibm.ivj.eab.j2cpp.PCHAR aCCardId = new
            COM.ibm.ivj.eab.j2cpp.PCHAR(aCardId);
        COM.ibm.ivj.eab.j2cpp.PCHAR aFileName = new
            COM.ibm.ivj.eab.j2cpp.PCHAR("C:/VAJResid/PPPCard/cards.txt");
        cardserver.J_CardManager aJ_CardManager = new
            cardserver.J_CardManager();
        aJ_CardManager.open(aFileName);
        COM.ibm.ivj.eab.j2cpp.PCHAR TheCard =
            aJ_CardManager.cardLine(aCCardId);
        aJ_Card = new cardserver.J_Card(TheCard);
        if ((String.valueOf(aJ_Card.cardNumber()).
            compareTo("NoCardFound")) == 0)
            fieldCardId = "Invalid Card";
        else {
            fieldCardId = aCardId;
            fieldCustomerId = String.valueOf(aJ_Card.customerId());
            fillTheAccounts();
            aJ_CardManager.close();
        } catch (java.lang.Throwable exc) {
            System.out.println("----- UNCAUGHT EXCEPTION -----");
            exc.printStackTrace(System.out);
        }
        return ;
    }
}
```

The `checkPin` method calls the `J_Card` to validate the PIN:

```
public boolean checkPin(String aPin) {
    /* Perform the checkPin method. */
    COM.ibm.ivj.eab.j2cpp.PCHAR aCPin = new
        COM.ibm.ivj.eab.j2cpp.PCHAR(aPin);
    if (aJ_Card.checkPin(aCPin) == 1) return true;
    else return false;
}
```

The private `fillTheAccounts` method calls the `J_Card` to retrieve the list of accounts into the accounts Vector:

```
private java.util.Vector fillTheAccounts () {
    fieldAccounts = new java.util.Vector();
    for (int i=0; i<10; i++) {
        java.lang.String anAccount =
            String.valueOf(aJ_Card.accounts(i));
        if (anAccount.compareTo("0")==0) break;
        fieldAccounts.addElement(anAccount);
    }
    return fieldAccounts;
}
```


Testing the C++ Card Server Bean

Before we integrate the CPPCard bean into the ATM application, we test it with a small applet called CardTest2 (Figure 182).

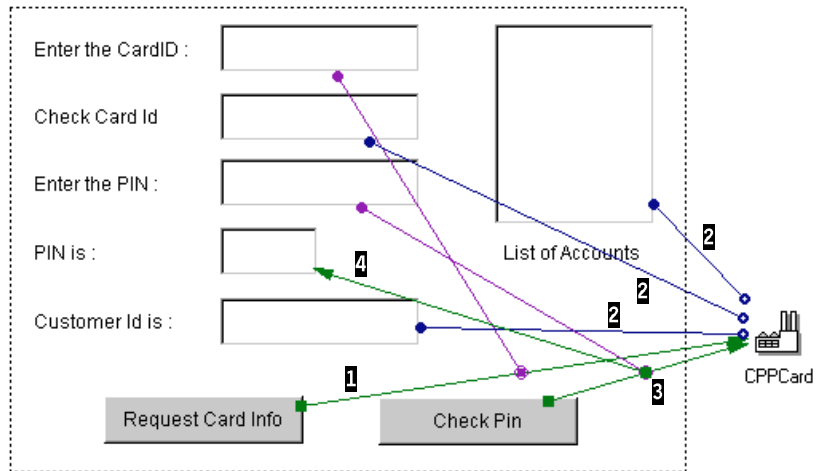


Figure 182. Testing the C++ Card Server

The **Request Card Info** button is connected to the CPPCard constructor, with the card ID field as a parameter (1). This action retrieves the customer ID and the list of accounts and displays the information in the panel through property connections (2).

The **Check Pin** button is connected to the checkPin method with the PIN field as a parameter (3), and the result of the validation is displayed in the panel (4).

Integrating the C++ Card Server into the ATM Application

The example developed here does not fit easily into the design implemented in the distributed ATM application:

- ❑ In the ATM application, PIN validation is done locally after the card information is retrieved from the DB2 server database; the C++ card server implements validation in the C++ code.
- ❑ The card object in the ATM application holds real account objects; the C++ card server holds account numbers only.

In the local ATM application we could replace the card bean with the C++ card server bean for PIN validation, but to make the rest of the application work we would have to construct a card object for the other panels and functions.

10

Access to VisualAge Generator Servers

In this chapter we introduce you to VisualAge Generator, explain how support for Java clients is provided as part of VisualAge Generator, and show how Java, combined with VisualAge Generator Java support, can play a significant role in the development and implementation of robust e-business solutions.

Detailed information about VisualAge Generator and Java clients can be found in *Unlimited Enterprise Access with Java and VisualAge Generator*, SG24-5246, to be published mid 1998.

VisualAge Generator Support for Java

The real challenge facing Java computing today is how to use Java to marry the advantages of electronic commerce to robust, scalable, transaction systems. After all, these transaction systems have been providing support for mission-critical application and data management processes for decades.

VisualAge Generator provides a seamless interface for Java programmers to extend existing transactions and rapidly deploy new transactions for Java clients.

We begin with a quick overview of VisualAge Generator and the Java support it provides.

VisualAge Generator

VisualAge Generator is a full-function rapid application development environment used to build and deploy multitier client/server application systems. VisualAge Generator can deliver a full range of application architectures, from stand-alone 3270 and batch application systems to GUI-based client/server application systems.

For client/server systems, VisualAge Generator builds high performance server programs that can be implemented natively on OS/2, Windows NT, OS/400, AIX, and HP-UX platforms as well as in transaction processing environments such as CICS and IMS. GUI and text clients use the VisualAge Generator PowerServer API (VisualAge Generator middleware) to easily connect server programs running on Windows and/or OS/2 (and now Java clients as well).

The VisualAge Generator programmer uses IBM's VisualAge "construction from parts" visual programming paradigm and the VisualAge Generator fourth generation language (4GL) to develop and test all tiers of the client/server application on the development workstation. The visual development environment is built on top of the same visual builder used by VisualAge Smalltalk, VisualAge C++, and VisualAge Java. The VisualAge Generator programmer codes business applications in this scripting language while receiving all the benefits of visual programming provided by VisualAge.

Application systems are developed with a focus on the required business logic and not the complexities of the target run-time environment or system configuration. VisualAge Generator improves the ease of development by:

- ❑ Generating all of the system-dependent code necessary for the functions implemented in a program for the chosen target run-time environment
- ❑ Implementing client/server communication, using the PowerServer API.

This isolation from the complexity of the target run-time environment and the implementation of intersystem and interprocess communications leaves the programmer free to focus on business logic. Removing the programmer from the system-dependent details of file and database access and client/server program-to-program communications is what makes VisualAge Generator a rapid application development tool.

The programmer has a wide range of development options. The application system can be constructed and entirely tested on a single development platform independent of the target run-time environment or client/server configuration. When ready, server programs can be deployed to the target run-time environment and then accessed by the development platform to support client program testing. This mix of pure source debugging and access to the run-time code for servers, when available, provides the programmer with a highly productive development environment.

Once the application system has been constructed and tested, the programmer generates a single set of source components for use in one or more of the client and/or server platforms shown in Figure 183.

★ **Multiple Client Platforms**

★ **Development Environment Tests Server Source or Calls Run-time Server through the PowerServer API**

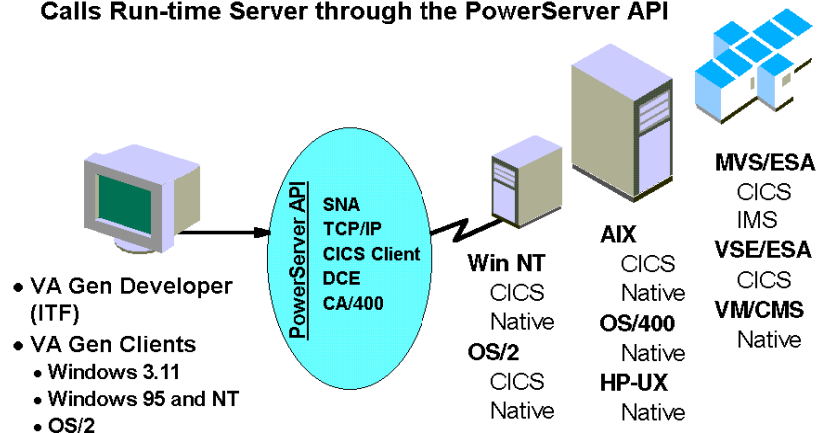


Figure 183. VisualAge Generator Client/Server Support

During the generation of application system components, the 4GL scripting language is transformed to COBOL or C++ as required for the target run-time platform.

Clients and servers communicate using the VisualAge Generator PowerServer API, which maps all client requests to the appropriate middleware option for the target server and run-time environment. For example, if the client is communicating with a server program on MVS CICS, the client call creates a CICS ECI block and passes the request to the CICS client/server programs that access DB2, DL/I, or indexed file data through a common I/O paradigm that is easy for programmers to understand.

VisualAge Generator Java Support

VisualAge Generator participates in multiple solutions for Web-based access to application systems:

- ❑ For existing 3270 text interfaces implemented in CICS, the CICS Internet Gateway can be used to provide distributed user interface access in a Web browser.
- ❑ VisualAge Generator GUI clients can make use of the HTML parts provided by VisualAge Smalltalk to implement Web browser access to Smalltalk and server program functions.
- ❑ VisualAge Generator can generate Java beans that wrap VisualAge Generator server programs to provide a direct connection between a Java client and a VisualAge Generator server program.

VisualAge Generator Java support provides direct access to server programs in a Java environment. Using the server definition as input, VisualAge Generator generates Java beans that *wrap* calls to server programs. You can import the Java beans generated by VisualAge Generator into your favorite Java development environment and develop applets or applications that *call* VisualAge Generator server programs.

In fact, once you have written the Java client, it can call the VisualAge Generator server on any of the supported run-time platforms. The Java client code used to call the VisualAge Generator server program is generic. The VisualAge Generator run-time configuration controls where the server will be (platform) and how it will be called (middleware implementation option). The same Java client could call a server program on CICS (MVS, VSE, AIX, NT or OS/2), IMS, OS/400, AIX, HP-UX, OS/2, and Windows NT (see Figure 184).

- ★ **Java Access to Scalable Transaction Servers**
- ★ **Rapidly Create NEW Transactions for Java Clients**

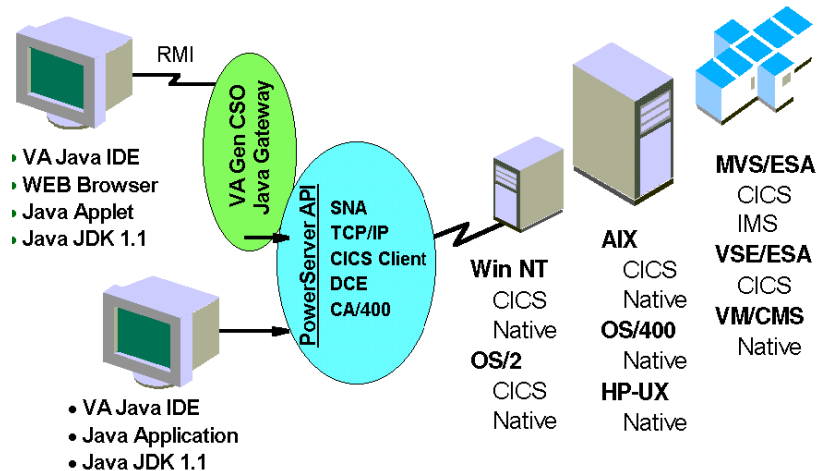


Figure 184. Java Client and VisualAge Generator Server Support

VisualAge Generator support for generating Java beans provides a very exciting and attractive way of providing an end-to-end Java transactional computing solution in an e-business environment.

The flexibility of Java client delivery, combined with robust, scalable servers that can be implemented in any of the target run-time platforms supported by VisualAge Generator, provides the Java programmer with a rapid application development (RAD) solution for the complete enterprise. Not only can you extend the legacy application systems that already exist on your enterprise servers, you can create new transactions on these servers for both your internal operational systems and your external (Web accessible) e-business computing solutions.

Implementation of Java Support

In this section we review the implementation of VisualAge Generator support for Java. Two key components provide support for Java client access to VisualAge Generator servers:

- ❑ VisualAge Generator CSO Java classes (PowerServer API enablement)
- ❑ VisualAge Generator generated Java beans (wrappers for servers and server parameters)

VisualAge Generator CSO Java Classes

The VisualAge Generator CSO Java classes are shipped with the VisualAge Generator Developer and VisualAge Generator Server products. These products provide development and run-time support for the OS/2, Windows NT, Windows 95, AIX, and HP-UX platforms.

The Java support currently shipped with VisualAge Generator enables Java applications and Java applets delivered by Web servers running on these systems to access the PowerServer API. The PowerServer API is the published server interface in a VisualAge Generator client/server environment. This server interface provides access to the function and middleware connectivity options provided by VisualAge Generator.

VisualAge Generator Generated Java Beans

VisualAge Generator provides access to VisualAge Generator server programs in a Java environment by generating Java beans that wrap the server programs and their parameters. These wrappers, when combined with the CSO Java classes, provide access to the VisualAge Generator PowerServer API.

The server program is developed in VisualAge Generator and generated for the chosen target run-time platform. A second generation step generates the Java beans required to access the server program from a Java client. These VisualAge Generator Java beans are then imported into your favorite Java development tool and used during the development of a Java applet or application.

The generated Java bean classes represent:

- ☐ Servers
- ☐ Record parameters
- ☐ Rows in substructured record arrays

Accessing VisualAge Generator Servers from Java Clients

The VisualAge Generator CSO Java classes and generated server and record parameter Java beans can be used in either a Java application or applet (an applet is running in a Web browser). The differences in how a VisualAge Generator server is called from an application or an applet are implemented and managed by the VisualAge Generator CSO Java classes.

From a Java Application

When a VisualAge Generator server Java bean is used in a Java application, the server bean calls the VisualAge Generator PowerServer API on the platform where the Java application has been implemented. Therefore VisualAge Generator CSO support (VisualAge Generator Common Services) must be installed and configured on the client workstation.

Figure 185 provides an overview of the complete process of implementing and using VisualAge Generator Java beans in a Java application client.

Java Application Client or Development Platform

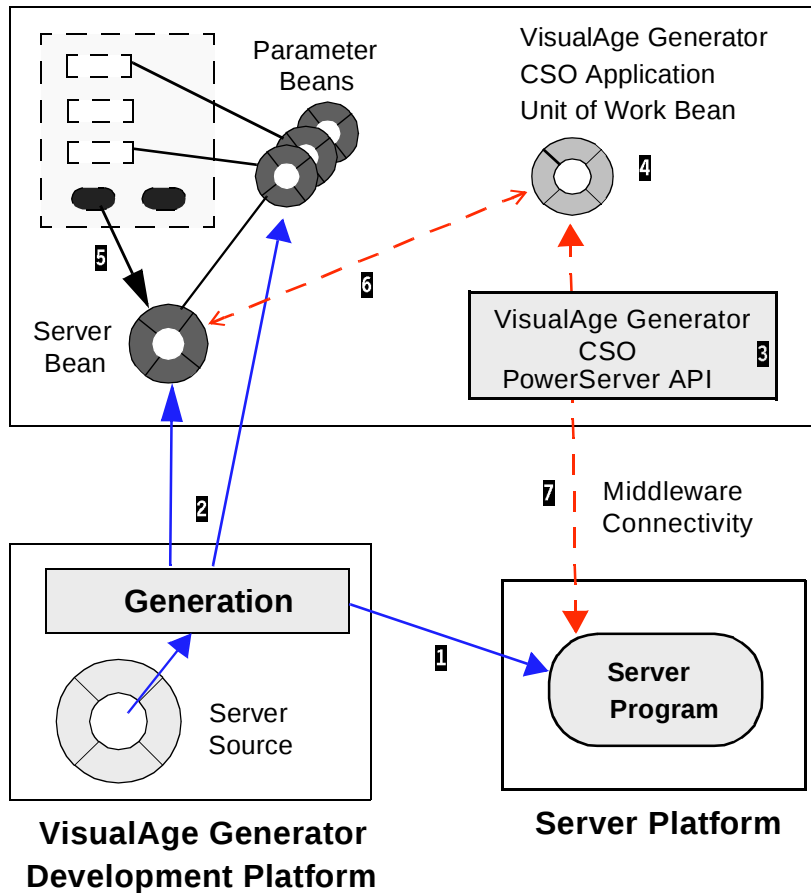


Figure 185. Developing a Java Application

Here are the steps depicted in Figure 185:

- ❑ VisualAge Generator has been used to develop a server program. The server is generated twice: once for the target run-time platform (1) and once to produce the Java beans for the server and record parameters (2).
- ❑ A Java application client run-time or development platform has been established. VisualAge Generator Common Services has been installed to enable the PowerServer API (3).
- ❑ The VisualAge Generator CSO classes and the generated Java beans have been imported into the VisualAge for Java workbench. A Java client is developed by using the generated Java beans and the VisualAge Generator CSO application unit of work bean (4).
- ❑ During Java client testing in the development environment (or during execution on a run-time platform), a request to call a VisualAge Generator server is triggered (5).
- ❑ The request to call the VisualAge Generator server program is implemented by both the generated VisualAge Generator Java bean for the server and the VisualAge Generator CSO application unit of work bean (6). Data is marshaled and converted (Unicode to ASCII or EBCDIC, as required), and the PowerServer API is accessed locally on the same platform as the Java client. The call is routed to the server platform through the middleware connectivity option specified in the VisualAge Generator client/server configuration (7).

From a Java Applet

When a VisualAge Generator server Java bean is used in a Java applet, the server wrapper runs on a Web client. When a server call is triggered, Java RMI processing is used to talk with the VisualAge Generator CSO gateway that is running on the Web server (that delivered the Java applet to the browser). The PowerServer API request for the server call is issued by the CSO gateway that was started on the Web server. Therefore VisualAge Generator CSO support (VisualAge Generator Common Services) must be installed and configured on the Web server that was used to load the Java applet.

Figure 186 provides an overview of the complete process of implementing and using VisualAge Generator Java beans in a Java applet client.

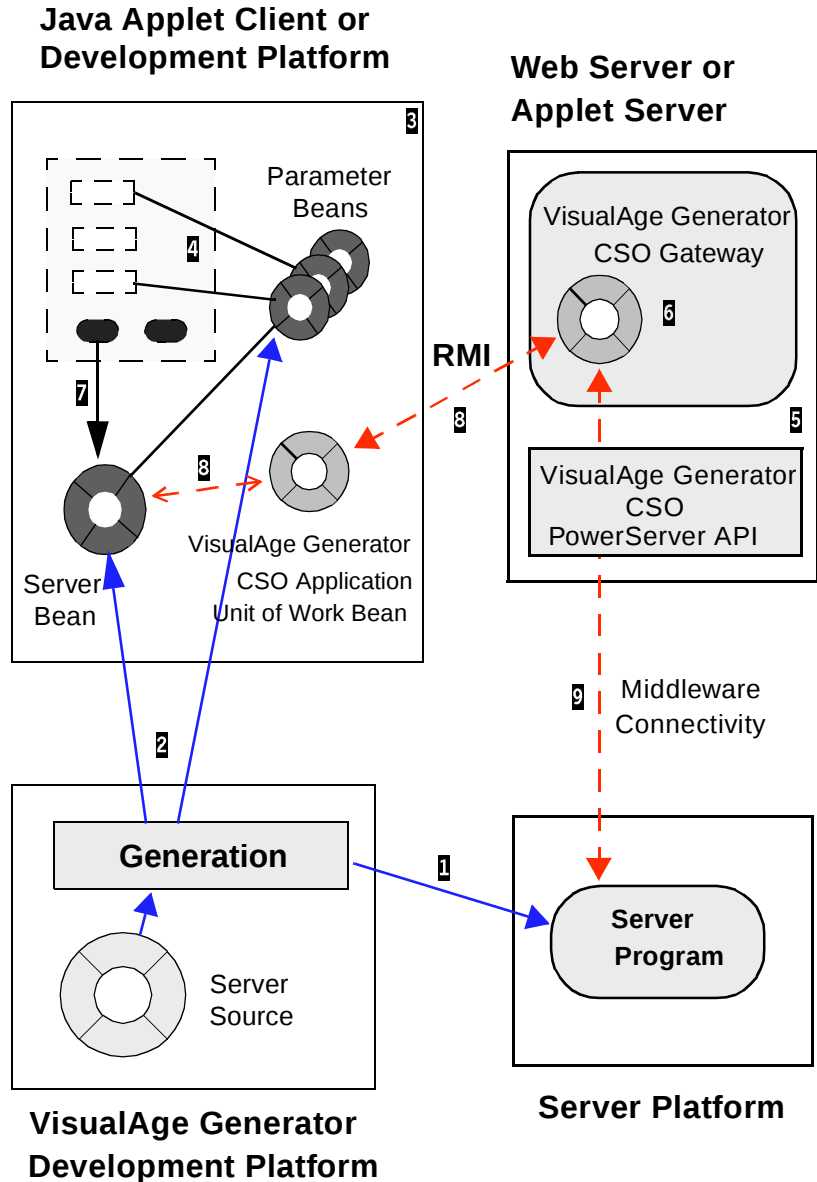


Figure 186. Developing a Java Applet

Here are the steps depicted in Figure 186:

- ❑ VisualAge Generator has been used to develop a server program. The server is generated twice: once for the target run-time platform (1) and once to produce the Java beans for the server and record parameters (2).
- ❑ A Java applet client run-time (applet viewer in a Web browser) or development platform has been established (3). VisualAge Generator Common Services is not required on this platform.
- ❑ The VisualAge Generator CSO classes and the generated Java beans have been imported into the VisualAge for Java workbench. A Java client is developed by using the generated Java beans and the VisualAge Generator CSO application unit of work bean (4).
- ❑ A Web server capable of delivering the Java applet (for a run-time Web browser client) is established. VisualAge Generator Common Services has been installed on the Web server to enable the PowerServer API and the implementation of the VisualAge Generator CSO Java gateway (CSO UnitOfWorkServer) (5).
- ❑ On the Web server platform the VisualAge Generator CSO Java gateway is started (6). This gateway responds to RMI requests from the Java applet client.
- ❑ During Java applet client testing in the development environment (or during execution on a run-time platform), a request to call a VisualAge Generator server is triggered (7).
- ❑ The request to call the VisualAge Generator server program is implemented by both the generated VisualAge Generator Java bean for the server and the VisualAge Generator CSO application unit of work bean. An RMI message is sent to the VisualAge Generator CSO Java gateway (8). At the gateway the data is marshaled and converted (Unicode to ASCII or EBCDIC, as required), and the PowerServer API is accessed on the same Web server platform as the gateway. The call is routed to the server platform through the middleware connectivity option specified in the VisualAge Generator client/server configuration (9).

The e-Business Solution: Java and VisualAge Generator

To appreciate the e-business solution provided by the combination of Java and VisualAge Generator, you need to understand how the solution works and why it is appropriate for your business demands.

Run-time Configuration for Java Applets and VisualAge Generator Servers

Adding VisualAge Generator to an e-business environment based on Web-enabled application systems that use Java is not that difficult. The Java beans are generated for you, and the implementation of RMI messaging between a Web browser and the CSO Java gateway has already been written. All that is left for you to do is configure a working e-business system as shown in Figure 187.

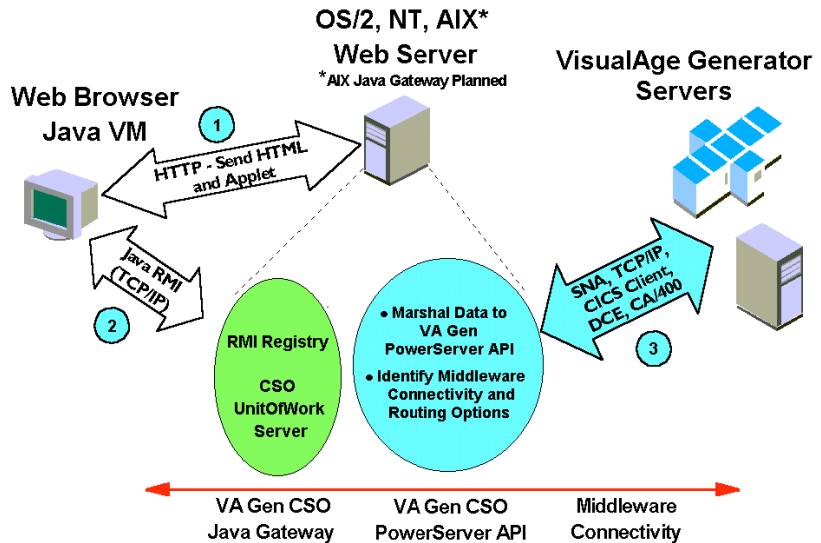


Figure 187. The e-Business Solution: Java Applets and VisualAge Generator Servers

As shown in Figure 187, the VisualAge Generator CSO Java gateway uses RMI to connect Java applets with the PowerServer API. This gateway provides you with a mechanism that extends the middleware connectivity options supported by a VisualAge Generator client/server communication configuration to Java clients.

As more platforms, such as AIX, and possibly MVS, are supported by the VisualAge Generator CSO Java gateway, additional configurations similar to that shown in Figure 187 will be available.

Value of VisualAge Generator in an e-Business Solution

Now that you understand what VisualAge Generator is capable of, and how easy it is to access VisualAge Generator servers from Java clients, it should be obvious that the best solution for network-enabled enterprise computing application systems (or ***robust e-business solutions***) is a marriage of Java programming technology for the client and VisualAge Generator programming technology for the server.

To support this view consider these attributes of a Java and VisualAge Generator e-business client/server solution:

- ❑ Java programming technology provides a common programming platform without concern for the run-time platform or operating system. In other words, Java clients can run on any Java-enabled platform or web browser.
- ❑ VisualAge Generator programming technology, as enabled by the generation function, provides a common programming platform without concern for the run-time platform or the operating system. In other words, VisualAge Generator servers can run on any of the run-time platforms supported by VisualAge Generator, that is, AIX, OS/2, Windows NT, HP-UX, IMS/ESA, and just about any CICS platform (MVS, VSE, Windows NT, OS/2, and AIX).
- ❑ Non-Java clients that might be required for internal operational systems can be developed with VisualAge Generator. These can be text (3270-style) client programs or full function GUI systems implemented using VisualAge Generator (which includes access to the full functions of VisualAge for Smalltalk).
- ❑ As demand on your server platform changes, VisualAge Generator generation technology can be used to move server functions to the appropriate run-time platform. With VisualAge Generator you can scale your server platform from a departmental solution (Windows NT) to an enterprise solution (AIX, CICS, IMS), without changing the Java clients! The Java clients are not affected by the change in the server run-time platform. VisualAge Generator and the PowerServer API automatically, and transparently, provide the appropriate data conversion and middleware connectivity support.

ATM Application with a VisualAge Generator Server

We implemented the ATM application, using a VisualAge for Java applet and VisualAge Generator servers.

The design of the application is based on the designs used in other chapters of this redbook:

- ❑ The server design is based on the design used in “ATM Application with the CICS Access Builder” on page 229.
- ❑ Access to the server from the GUI is based on encapsulating the server into Java beans, similar to the design used in Chapter 7, “ATM Application with RMI,” on page 171.
- ❑ The GUI uses individual frame windows that look similar to the panels used in “User Interface Classes” on page 115 for the JDBC implementation or in “View Layer” on page 202 for the RMI implementation.

Figure 188 shows the configuration used to support development and production. The same applet was tested with development servers on Windows NT and production servers under CICS on Windows NT.

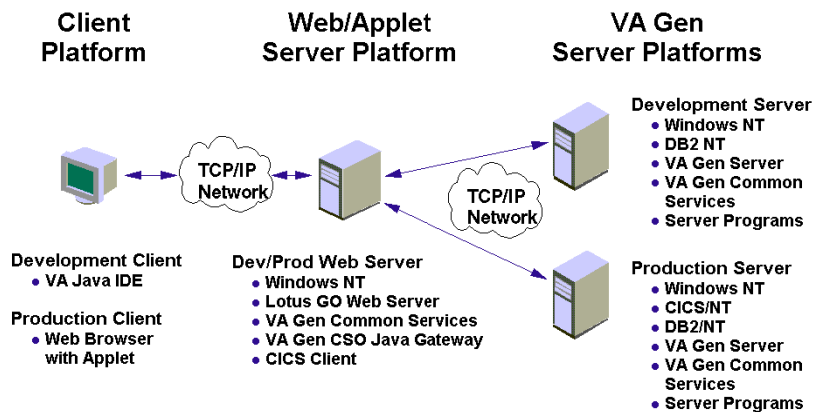


Figure 188. Development and Production Environment Configuration

We do not describe the implementation of the ATM application in detail in this redbook. Detailed description will be available in the redbook *Unlimited Enterprise Access with Java and VisualAge Generator*, SG24-5246, to be published mid 1998.

11

Access to Distributed CORBA Objects

In previous chapters we describe how VisualAge for Java helps you access enterprise data and business logic directly or through RMI. However, for a large enterprise intranet deployment, you face certain challenges, regardless of the architectural approach you choose. The challenges are security, scalability, concurrent access to resources, multiplatform support, and application management.

The CORBA standard architecture defines the specifications for interoperability and its associated services, and IBM provides an implementation through its Component Broker. In this chapter we describe how this environment can be used with VisualAge for Java.

The Case for CORBA

Figure 189 shows the various ways of accessing enterprise resources from Java. We have covered three of them so far. Now we discuss why we should consider a distributed CORBA object environment.

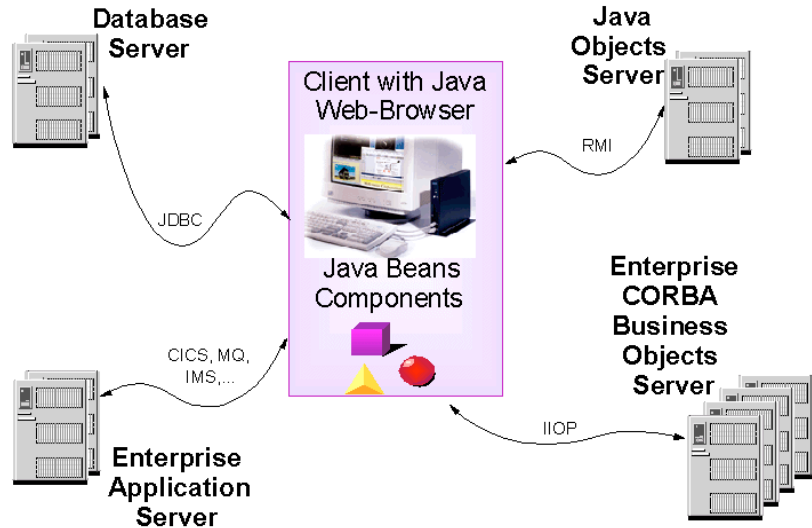


Figure 189. Enterprise Access from a Java Client

Why CORBA?

Because of mergers and acquisitions, companies are faced with disparate or duplicate solutions and systems that make distributed and heterogeneous computing over a network a requirement. Although groups with different business requirements have different set of systems and tools (NT, UNIX, mainframes, COBOL, 4GLs Oracle, DB2, CICS), nevertheless it makes sense for business analysts to incorporate customer service trends directly into their analyses, or for accounting to incorporate marketing projections directly into accounting budget spreadsheets.

With the emergence of the Web, there is a strong push to provide ubiquitous access from dealers and customers through the Internet, in addition to corporate intranet access, with an integrated view of many servers. The Web architecture replaces the traditional client/server architecture with leaner clients. This shift to Web-based technologies forces IT organizations to make transitions from old to new that are not easy and offer benefits that outweigh the cost of replacement or modification. Twenty or thirty

years of development cannot be thrown away; previous large investments in CICS or IMS transactional services have to be preserved. Existing data, business logic, and resource managers from many vendors must be leveraged. Figure 190 shows an example of the integrated view of services a bank is expect to provide, hiding the variety of systems.

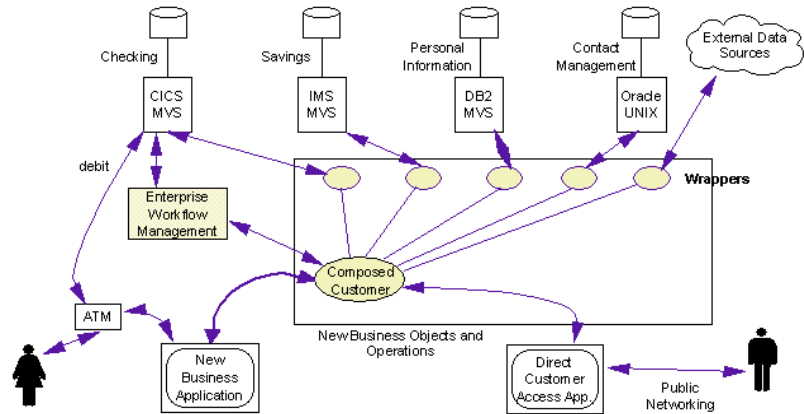


Figure 190. The Enterprise Distributed Environment

The traditional client/server approach (two-tier architecture) lets client applications written in 3GL or 4GL languages access relational databases with SQL and stored procedures or CICS transaction services. Business functions are implemented in a language-, middleware-, network-, and operating-system-dependent manner, often in the client-side application space. Such a fat client deployment has become a nightmare to distribute and manage. Some companies also have developed their own middleware.

How can information technology ensure a better evolution and integration? Building applications from components has tremendous appeal because it provides a natural way to encapsulate business functions. Starting everything from scratch with object orientation can be an overwhelmingly difficult task. Incremental development is necessary, leaving the system as is, improving the interface, integrating the existing functions, and adding new functions.

Tools such as the VisualAge family of products can reduce software development costs, using rapid prototyping and visual development environments. We have seen with VisualAge for Java Enterprise that business logic on servers can be reused, providing flexible interfaces to existing (legacy) applications. Such a strategy also allows the incorporation of new technology and promotes new styles of applications.

For some applications it may be sufficient, but, for building a global intranet and extranet, a common infrastructure, relying on a standard, open architecture as a corporate backbone is often necessary to incorporate and manage all these components. This leads to a logical three-tier distributed application model as shown in Figure 191. This three-tier model, which is appropriate for meeting business needs, can be consolidated in a two-tier physical environment, by using large, central servers for reduced operations and maintenance costs.

Another factor to consider is that the client side evolves very quickly, every Web year (three months!), and the server side evolves more slowly. A middle tier could also provide a buffer that isolates the evolution of these two tiers.

"An Application Architecture for the Enterprise"

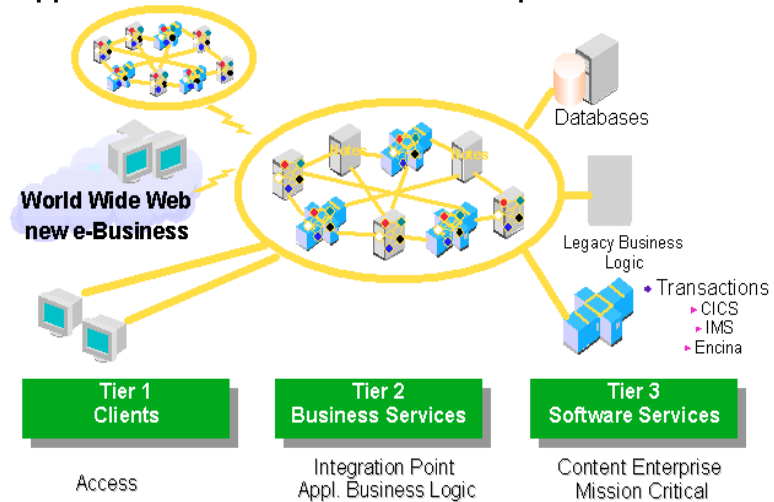


Figure 191. Three-Tier Architecture

Java has been the answer to providing an object implementation independent of platform and operating system, and CORBA is the answer to providing a complete, standard, distributed object infrastructure. Facilities provided by the relational database or the transaction system have to be extended at the distributed object level. These facilities include:

- ☐ Creating, locating, and sharing objects
- ☐ Querying object collections
- ☐ Ensuring authentication and authorized access to objects
- ☐ Preventing conflicting use of shared resources (with locking, accounting of deadlock, race conditions, and performance issues)

- ❑ Managing thread pools for servicing multiple client requests
- ❑ Choosing the caching policy on a server or client
- ❑ Ensuring scalability (System throughput should be able to grow to accommodate new demands and users, efficient use of resources, adding new servers transparently, and load balancing.)

Management of applications is also a critical piece of a robust deployment environment and includes performance tuning, load balancing, dynamic installation of new services, and quality of service monitoring.

What Is CORBA?

Let us now look at the origins of CORBA.

Object Management Group

The Object Management Group (OMG), established in 1989, is an open consortium of more than 800 companies that work together to define open standards for an architectural framework for object computing: CORBA/IIOP, Object Services, Internet Facilities, and Domain Interface specifications. The first publication of the OMG's Object Management Architecture Guide (OMA) dates from 1990.

CORBA 1.1 was introduced in 1991 and defined the Interface Definition Language (IDL) that enables client/server object interaction within a specific implementation of an object bus, called an Object Request Broker (ORB). CORBA 2.0, adopted in December 1994, defines true interoperability by specifying how ORBs from different vendors can interoperate.

This architecture has four main elements, as shown in Figure 192:

- ❑ The Object Request Broker for objects to intercommunicate
- ❑ CORBA Object Services (COS) that define system-level services that are added on to the ORB, such as security, naming, and transaction
- ❑ CORBA facilities, which define application-level services, such as compound documents and other vertical facilities
- ❑ Application (or business) objects, which describe real-world objects and applications, such as an airplane or a bank account

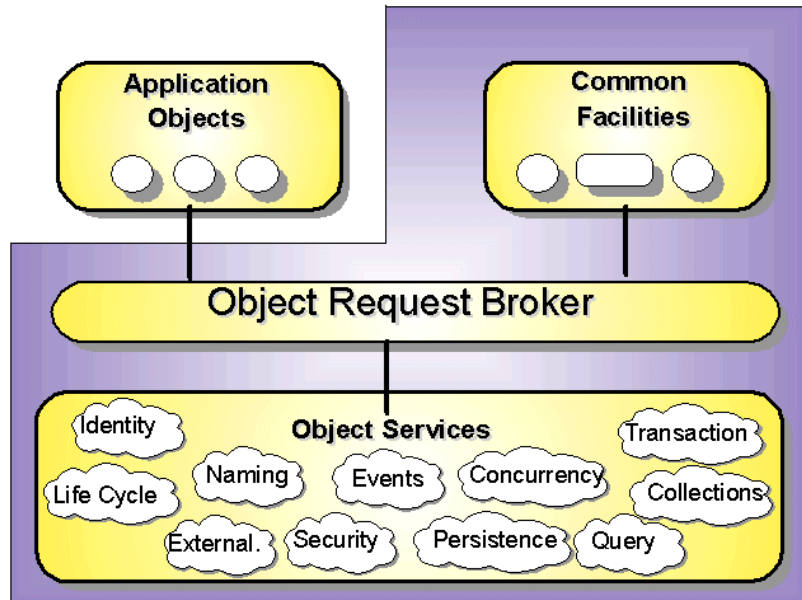


Figure 192. The Object Management Architecture

Object Request Broker

The ORB is the middleware that establishes the client/server relationship between objects. The ORB intercepts the call and is responsible for finding an object that can implement the request, pass it the parameters, invoke its method, and return the results. The client does not have to be aware of where the object is located, its programming language, its operating system, or any other system aspects that are not part of an object's interface.

The ORB is the only portion of CORBA that must be present in order to build a CORBA-compliant application. Many ORBs ship without any of the CORBA services or facilities, and you must create (or purchase) these services yourself. However, without the ORB, a CORBA application cannot function. The most visible function of a CORBA ORB is to respond to requests from your application or from another ORB. During the life-cycle of a CORBA application, the ORB may be asked to do many different things, including:

- ☐ Look up and instantiate objects on remote machines
- ☐ Marshal parameters from one programming language (such as Java) to another language (such as C++)
- ☐ Invoke methods on a remote object, using the static method invocation described by a downloaded stub

- ❑ Invoke methods on a remote object, using dynamic method invocation
- ❑ Automatically start objects that are not up and running
- ❑ Route callback methods to the appropriate local object that it is managing

The great thing about the ORB is that nearly all of the implementation details for all of these duties are hidden from the software developer. Simply providing the appropriate “hooks” in your code to initialize the ORB and register your application with the ORB opens your application up to a vast galaxy of distributed objects.

Interface Definition Language

The key to language indulgence is the IDL. This wrapper exposes the services the application can offer and provides a standard interface for initiating methods from clients. CORBA objects can be written in any programming language that a CORBA software manufacturer supports, such as C, C++, Java, or Smalltalk. As other languages grow in popularity, CORBA vendors undoubtedly will release bindings for those languages as well. The CORBA objects can exist on any platform that a CORBA software manufacturer supports, such as Windows 95 and NT, OS/2, AIX, Solaris, and MVS.

IDL is a descriptive language, with a syntax close to C++. Just as properties and methods are grouped together into classes in Java, these items are contained within interfaces in IDL. A module can group one or more interfaces, just as Java packages group classes. An operation is the equivalent of a Java method, with its signature, that is, parameters and return type. A parameter is defined as *in*, when the value is passed from client to server, *out* from server to client, and *inout* for both directions. These operations can raise exceptions. CORBA data types are defined and map to native data types through the appropriate language bindings.

The following code snippet shows a simple IDL module with a basic interface (we expand on this code in “Java Client Accessing a CBConnector Server” on page 337):

```
module MyBank
{
    interface Account
    {
        attribute string accountNumber;
        exception anException {string reason;};
        void changeBalance(in double anAmount)
            raises (anException);
    };
};
```

When we compile this IDL module using an IDL-to-Java compiler, we get a Java-language-specific stub and skeleton. The client IDL stub defines how clients invoke corresponding services on the server; a client has one IDL stub per interface it uses on the server. This stub performs marshaling. It also includes header files that enable you to call methods on the server from the chosen language (C++, Smalltalk, Java). The server skeleton provides interfaces to each service as well as demarshaling. The interfaces are registered in the ORB Interface Repository. Clients can invoke the methods through the ORB without ever knowing that the implementation is in a legacy environment.

The Object Adapter provides the run-time environment for instantiating server objects, passing requests to them, and assigning CORBA object references (IDs). These IDs are registered in the server implementation repository. CORBA defines interoperable object references (IORs) that each vendor must use to pass object references across heterogeneous ORBs. Figure 193 shows how all these elements interoperate.

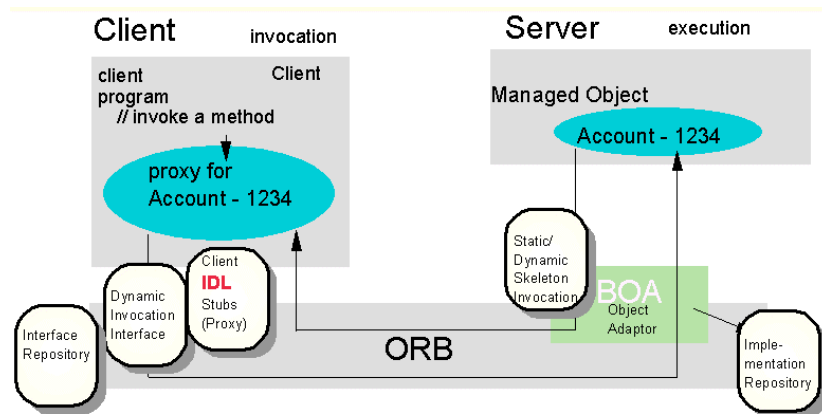


Figure 193. Client/Server Invocation with IDL

CORBA also has a dynamic invocation API that allows a client program to dynamically build requests on an object. It provides maximum flexibility but is less efficient. It is primarily used in tools, and we do not cover it further in this book.

IIOP Communication Protocol

Because CORBA 1.1 completely left the implementation of the ORBs to the vendors, components could not properly interoperate across various ORBs. CORBA 2.0 added interoperability by specifying a mandatory Internet Inter-ORB Protocol (IIOP). IIOP is a particular implementation of the General Inter-ORB Protocol (GIOP) over TCP/IP.

GIOP specifies message formats and common data representation (CDR) so that two ORBs can communicate with each other. In particular CDR takes care of different data representations across heterogeneous platforms.

GIOP defines the format for IORs. An IOR is created from an object reference and used whenever an object is referenced across an ORB, in order to find the object.

CORBA Services

CORBA services are a collection of system-level services that you can integrate with the business components through the IDL. If you want to develop an account that is persistent, transactional, and can manage concurrency, the interface must use multiple inheritance or add before and after callbacks to those services.

Naming Service

The naming service is one of the basic services because when objects are first started they need to find other objects. The naming service maps human readable names to IOR by a mechanism called *name binding*.

To help you manage a large number of distributed objects, the naming service allows you to put objects into a hierarchy of contexts. Each context groups a set of related objects just as a directory or a folder in an ordinary file system stores related files. For example, you can put all objects executing the same application or all objects providing the same type of services in the same context.

Life Cycle Service

To create a new object, a client must find a factory object that knows how to instantiate an object of that class and get back a reference. The life cycle service provides object creation, as well as an interface for copying, moving, and deleting existing objects.

Security Service

The security service protects critical resources from intentional or unintentional misuse:

- ❑ Authentication makes sure that you are who you claim to be when you use a service. This aspect guarantees that servers can trust their clients.
- ❑ When a client is authenticated, the server uses access control lists (ACLs) to ensure that the client is allowed to access resources.
- ❑ No client or server can deny having been involved in a communication, that is, the sender has a proof of delivery, and the receiver has proof of the server's identity. This is called *non-repudiation*.
- ❑ The data is securely transmitted (with encryption) and checked for corruption (checksum).
- ❑ Audit services allow system managers to monitor activities.

Event Service

The event service allows processes and objects to be notified when particular events occur during the execution of a method or application. Different parties are involved in the communication between objects through events.

The event service decouples the communication between event suppliers and event consumers. The event suppliers produce event data, and the event consumers process event data. Event data is communicated between event suppliers and event consumers through standard CORBA requests.

Suppliers do not need to know about consumers interested in the events they generate. Consumers do not necessarily need to know in which object a particular event occurred. They only need to inform the event service about their interest in specific events.

There are two approaches to initiate event communication: the push model and the pull model. The push model allows a supplier of events to initiate the transfer of the event data to the event consumer. The pull model allows a consumer of events to request the event data from the event supplier.

An event channel is an intervening object that allows multiple event suppliers to communicate with multiple consumers asynchronously. An event channel is both an event consumer and an event supplier.

Identity Service

In a distributed system, a task as simple as checking whether two references point to the same object needs special handling. Only the identity service, with specific methods such as `is_identical`, can safely indicate whether or not two different object references represent the same real object.

Externalization Service

The externalization service has two main purposes:

- ❑ Create a new object with the state of another object and move or copy the state of objects to different memory locations between hosts
- ❑ Move the state of objects in and out of a persistent store

This is similar to the serialization mechanism of Java.

A standard stream data format defines how to exchange streams across different operating systems.

Transaction Service

One of the most important components of any distributed environment is the object transaction service (OTS). To provide robustness in a distributed system, participants at different locations of the system should work together in a coordinated fashion. The OTS provides this coordination. We have already seen in “Host CICS Access with the CICS Access Builder” on page 221 that CICS provides a transaction environment for procedural applications. In the same way, OTS defines when a transaction starts as a unit of work from a client, all the servers involved, how it ends successfully, or how it recovers in case of failure (with commit or rollback). OTS also defines a two-phase commit protocol to coordinate a commit or abort of transactions across multiple servers so that they all fail or succeed.

OTS defines four key interfaces—`Current`, `TransactionalObject`, `Coordinator`, and `Resource`—that the client object can use. For instance, in the case of a transfer of money from one account to another account:

- ❑ `Account` inherits from `Resource` and `TransactionalObject` interfaces; this makes the account recoverable.
- ❑ The client invokes a `begin` on the `Current` object to maintain an active transaction.

- ❑ The client calls the debit and credit methods implemented in the account object. Because accounts are recoverable objects, they use services from the Current and Coordinator objects.
- ❑ When the client issues the commit, the commit method of the Current object is invoked, and the Coordinator performs the two-phase commit. The operation fails or succeeds completely.

Concurrency Service

The concurrency service provides a means of coordinating access to resources. When several clients try to use a resource, the clients will be serialized in a way that keeps the resource in a consistent state. The concurrency service is intended primarily for use in a transactional environment, where locks can be acquired on behalf of the transaction. When the transaction ends, those locks are freed. You can use the concurrency service to control access to resources without transactions. In this case, you have to release the locks yourself at the appropriate time.

Query Service

The query service provides predicate-based query operations on collections of objects. Queries can be specified using object derivatives of SQL and/or other styles of object query languages, including direct manipulation query languages.

The term “query” has read-only connotations, but we use it to denote general manipulation operations including selection, insertion, update, and deletion on collections of objects.

Persistence Object Service

The persistence object service (POS) allows the state of an object to be persistent across multiple instantiations. The object can be saved in a data store and recovered when it is needed. POS can handle multiple storage mechanisms to object databases as well as relational databases or flat file systems. A persistent object must inherit from some interfaces defined in POS in order to get a mechanism for externalizing its state.

Other Services

Multiple other services are defined today (17 in total), and more are expected to be standardized or improved through the OMG request for proposal (RFP) mechanism. All those services enrich the behavior of components and provide a robust environment in which they can thrive.

CORBA and RMI

We have seen that RMI:

- ❑ Allows a Java client to instantiate objects that may be located on a remote server
- ❑ Interacts with remote objects through their published interface

But RMI has limitations for building a complete three-tier environment:

- ❑ The server application must be written in Java.
- ❑ RMI does not include any services to manage server objects.
- ❑ RMI provides only a limited, nonpersistent naming service to let you locate remote objects. You can look up only instantiated objects registered with the RMI registry. Unlike CORBA, an RMI Java client cannot create an instance on a server.
- ❑ RMI does not support self-describing objects, dynamic invocations, and interface repositories (a limitation of lesser importance in most enterprise applications).

However:

- ❑ RMI uses a reference-counting garbage-collection scheme that keeps track of all external live references to server objects, that is, an active client/server connection. As long as the reference exists, the remote object cannot be garbage collected (but a broken link can cause undesired server object garbage collection).
- ❑ With RMI you do not have to deal with CORBA IDL or with Java-to-CORBA type translations.

In short, RMI is a viable option for smaller-scale applications to be written entirely in Java. CORBA provides the foundation for integrating existing objects with new code and the ability to scale for the future through its services, and Sun has publicly stated its long term goal to allow RMI objects to communicate through IIOP.

How Java Complements CORBA

Java and CORBA fit well together because Java has a strong component model, Java beans, and CORBA treats such objects as well-defined services with standard calling interfaces. Java and CORBA also complement and strengthen each other, and CORBA is a good solution for deploying and managing distributed Java objects. Java emphasizes implementation transparency, and CORBA provides network transparency and services.

Therefore, it is a good idea to build a solution on top of these two pillars, as we explain in “Java Client Accessing a CBConnector Server” on page 337. But first let us describe how the IBM Component Broker implements the CORBA specifications.

Component Broker

Component Broker is IBM’s CORBA implementation. It provides a middleware solution for distributed object computing in a multitier environment. It includes a CORBA 2.0-compliant ORB that permits a variety of clients to collaborate with business objects residing in multiple servers. It is aimed at highly scalable, robust environments. Business objects can be generated by modeling tools. Their manageability is tool-generated by inclusion inside a framework. Their essential state is provided through tool-generated data objects that map to various back-end systems, data sources, and transactions (DB2, CICS, IMS). Therefore, Component Broker coexists with existing environments and facilitates the transition to component-based programming. Configurable CORBA services are integrated in the overall solution, which includes a development, run-time, and systems management dimension.

These features are grouped in two elements, CBConnector and CB Toolkit.

CBConnector

CBConnector is the application run-time environment, providing

- ❑ Execution
- ❑ Integration and management of object services, resource allocation, and unit of work definition
- ❑ Delivery, installation, monitoring

CB Toolkit

CB Toolkit is a set of development tools that enable developers to add run-time policy and deployment infrastructure to the basic business logic. It also enables the generation of proxies for inclusion in Java Web clients, C++ clients, and Active X clients.

VisualAge for Java, on top of its intrinsic functions, complements Component Broker by allowing the creation of Java applets and applications that run on the first-tier clients that make use of the generated server objects.

Run-time Architecture Components

Component Broker provides a CORBA 2.0 compliant server ORB, written in C++ for efficiency, which supports C++ server objects and allows for in-process efficient interaction between C++ and Java server objects. To easily accommodate Java, the object model does not use multiple implementation inheritance.

Programming Model

In the Component Broker programming model, the business logic has a number of visible client interfaces and an implementation executing in the CBConnector run-time environment according to the OMG CORBA architecture. The Component Broker object model provides memory and thread management, serialization, exception handling, and other operations, inside a framework.

The programming model defines:

- ❑ The set of tasks to be completed
- ❑ The set of roles (for example, connect the object model to the relational database tables)
- ❑ A set of classes to extend (for example, to participate in a transaction-extended framework with methods to implement, called by the transaction engine)

The programming model is provided through a Managed Object Framework (MOFW) and Application Adapter Frameworks.

Managed Object Framework

CBConnector cleanly separates the concerns of the client programmer from those of the business object implementer and the provider of the system plumbing. One very important consequence of this strategy is that you can run the same server-side business logic on top of widely varying back ends with the quality of service that they provide.

The MOFW is the core of the Component Broker programming model. Frameworks by themselves usually do not constitute an executable environment. Rather, they exist as a set of configurable and extensible classes that are adapted to the case at hand.

A Component Broker component completes the MOFW by implementing framework methods that enable components to be managed. It is completed by Application Adapter Frameworks that we describe in the next section. Figure 194 shows the features of the Component Broker Managed Execution Environment.

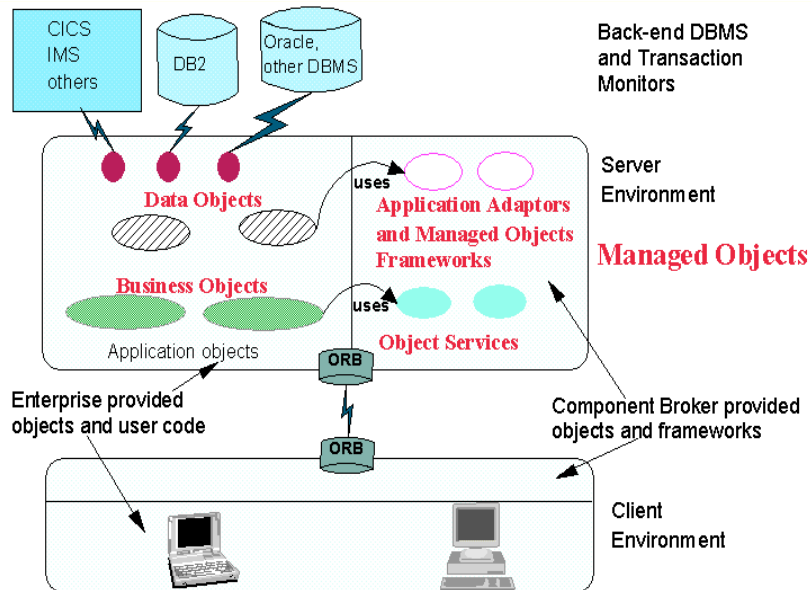


Figure 194. Component Broker Managed Execution Environment

The business object represents the business logic interface and implementation. A business object can have state objects, called data objects, that map business object attributes to different back ends. Data objects, using a caching technique, take care of synchronizing data with database systems. A data object might also retrieve the information needed by a business object through invocation of a back-end CICS transaction program.

A business object is assigned to a container to make it a manageable object. An managed object inherits from a business object and implements interfaces of its container. Most operations are not directly implemented within a managed object but instead are delegated to methods in another object known as a *mixin* object, so that no multiple implementation inheritance is required. The mixin object implements interfaces in components such as CORBA object services or the CBConnector Application Adapter Framework. The good news is that you generally do not have to concern yourself with framework-provided plumbing. It is pulled in for free, and the necessary code is generated through the CB Toolkit, in particular the Object Builder.

As explained in “Systems Management” on page 336, CBConnector does not forget the systems management dimension; what is generated (or already present within the CBConnector frameworks) contains all the necessary hooks for configuring it at installation time and controlling it during run-time.

Application Adapter Framework

While you design and code the business part of the objects managed by a CBConnector server, the Application Adapter Framework supplies the environment within which aspects such as an object’s transactional, persistence, or security characteristics are handled. In true framework fashion, the Application Adapter Framework invokes methods within your code at the appropriate time, for example, when it is time to write information out to persistent storage.

Application adapters map external data and transaction services to components so that they allow component-based applications to interoperate with existing procedural applications that use the same back-end systems. The first application adapter that is provided is for DB2, but IBM and other vendors will provide adaptors for resource managers such as CICS, IMS, MQSeries, Oracle, and SAP.

Application adapters consist of:

- ❑ Containers for objects of similar environments and allow for operations like create, update, retrieve, and delete of instances. They bring a quality of service to managed objects as well as policies for passivation, caching, and locking.
- ❑ Homes, the “birthplace” of your objects. In other words, you use homes to create business objects. Homes give you a collection-type interface that allows you to iterate over the objects within the home.

Object Services

Based on the COS specifications, CBConnector provides for object services required for distributed applications including creating, identifying, and controlling access to components, ensuring that updates are atomic, and maintaining state information. This is the most complete implementation of COS on the market today, in terms of number of services, depth of functions, integration, and inherent management. It currently supports 9 of the 17 services (naming, security, events, identity, life cycle, externalization, transaction, concurrency, and query).

Workload Management

Large-scale, commercial application environments clearly need transaction processing (TP) monitor functions for scalable performance and high availability. Object Application and Transaction Monitor (OATM) brings these capabilities to the world of composed business objects.

OATM concentrates and dynamically dispatches large numbers of client connections into servers that can service those requests. Multiple servers present a single, logical image of a server group to clients. The decision that determines which server will execute a method is based on so-called bind policies defined through systems management facilities. Server groups provide not only a load balancing feature but also improved availability (for instance, by avoiding servers that become unavailable).

Client Enablement

Component Broker is designed to facilitate access from most popular types of clients in distributed application scenarios. Microsoft Windows and OS/2 desktops, and Internet and intranet browser clients are supported.

ActiveX and CORBA-compliant C++ client applications can use CBConnector server objects without further adaptation. ActiveX clients can be developed with popular tools such as Visual Basic and Visual C++. Java applets and Java applications are supported for Internet and intranet access.

Proxies are used to support client use of server-side objects. As we see in the next section, CBConnector provides all the facilities to generate proxies. Client proxies do not implement any business logic themselves; they forward requests to their server counterparts for execution.

The code for this system infrastructure gets to the client in a variety of ways. It can be downloaded from a Web server, as in the case of a Java applet running inside a Web browser, or it can be installed through the management facilities of CBConnector.

A C++ client ORB, a client Java ORB, and an ORB translating Microsoft Component Object Model (COM) interface calls into object requests on IIOP are shipped with Component Broker for distribution to client systems.

We explain in “Java Client Accessing a CBConnector Server” on page 337 how Java beans generated by CB Toolkit are used to build a client application.

Developing Distributed Object Applications with Component Broker

Developing a large-scale enterprise-level application involves different tasks that are accomplished by various groups of people:

- ❑ Object developers write business components on the basis of their business domain knowledge.
- ❑ System developers build system infrastructures, implementing specific qualities of service.
- ❑ Application assemblers integrate all these elements to meet the requirements

We explain in this section how the work of object developer can be integrated in the system infrastructure provided by CBConnector and automatically provide the persistence, recoverability, manageability, and quality of service to the business model.

Modeling, Analysis, and Design

Component Broker allows object models that have been developed through industry leading tools such as Rational Rose to be imported for implementation. Once a model has been imported, it can be further developed over time. ERWin by Logitech is another example of a tool that can be used to reengineer relational database management system tables into the CBConnector realm.

Object Builder

The Object Builder is specifically designed to build server objects. Because it understands the CBConnector framework architecture, it can generate most of the code needed to make the application you produce work within the CBConnector frameworks. This approach is called programming by framework completion.

SmartGuides generate skeletons that make use of the MOFW so that the complexity of accessing standard CORBA services and CBConnector management facilities is masked to enhance productivity.

Support for persistent data and legacy systems is also generated by Object Builder. If access to a data source is necessary for certain business objects, the Object Builder facilitates the mapping of relational database tables to the data object. Mappings for transactions (screen definitions and COMMAREA structures) are also enabled through SmartGuides. You will find similarities between the SmartGuides of Object Builder and VisualAge for Java Enterprise.

Edit, Compile, and Debug

The IBM VisualAge family of products complements CBConnector tools for these tasks. However, nothing prevents you from working with other widely used edit-compile-debug (ECD) environments. CB Toolkit also provides you with a remote debug and test environment.

You can also use IBM VisualAge for C++ and VisualAge for Java or other development environments for constructing your end-user interface.

Systems Management

It is one thing to develop robust applications by using a mature set of tools, but is quite another matter to deploy software for a large-scale production system network and keep it up and running. CBConnector has been built from the start with reliability, availability, and serviceability (RAS) and systems management in mind. The instrumentation hooks for vital management information have been architected into the product. Through these, CBConnector ensures that error reports, trace information, and performance data can be collected and sent. CBConnector systems management provides you with the ability to configure servers and server groups, deploy software, and monitor and control all aspects of a CBConnector network.

Java Client Accessing a CBConnector Server

Now it is time to do some real work again. We are going to see how easy it is to use VisualAge for Java to deal with CBConnector server objects.

Because the purpose of this book is to describe how to reuse enterprise business logic, we are going to reuse existing objects. These objects have been developed and described in *CBConnector Cookbook Volume 1*, SG24-2033, which teaches you how to use CBConnector to develop distributed object CORBA applications.

We do not reimplement the ATM application but rather describe how to build a Java client that reuses an account object running on a CBConnector server.

Account Interface Definition

In CBConnector, we deal with CORBA objects described in IDL. For our purpose, we are reusing an account object defined in Rational Rose and imported into the CBConnector object builder. This operation produces the following IDL:

```
#ifndef _Account_idl
#define _Account_idl
// Generated from Account.idl
// on 11/12/97 17:56:31
// by Object Builder
#include "IManagedClient.idl"
interface Account : IManagedClient::IManageable
{
    exception NotEnough { }; // end exception NotEnough
    readonly attribute string<10> accountNumber;
    attribute string<30> accountHolder;
    double getBalance( );
    void changeBalance(in double anAmount ) raises (Account::NotEnough);
}; // end interface Account
#endif
```

This Account definition represents a very simplistic banking account that has two attributes, `accountNumber` and `accountHolder`, both strings.

Two actions are possible on an account object: retrieve its current balance through the `getBalance` method, and modify the balance through `changeBalance`. This last operation returns a `NotEnough` exception if the balance goes below zero.

The use of the account object is trivial, and we are going to focus on how to create the necessary environment on the client side to access the CORBA world.

Account Development with CBConnector

Our goal in this document is not to show you how to develop the server side. Therefore, we assume that someone has done that job and has given us two IDL files: `Account.idl`, which describes our business object, and `AccountKey.idl`, which uniquely identifies account instances.

Now that the server side is done, we can start developing the client side.

Java Client

In this section we deal with the specifics of using a Java client to access managed CORBA objects on a CBConnector server.

Preparing VisualAge for Java

Before developing our client with VisualAge for Java, we have to do some work.

The first step consists of importing the CORBA Java ORB classes (provided in a large file called `somajor.zip`) into VisualAge for Java. This is easy to do but requires about one hour on a Pentium 166 MHz! After this import, you have to import a second small file (`somojod.zip`) that contains a missing security class.

The second step is to generate a few Java files containing classes that represent our objects on the server and are commonly called proxies.

Generating Java Proxies

The first way to generate a proxy is by instructing the CBConnector Object Builder to generate a Java client proxy for the managed account object.

The second way to generate a proxy is to use the IDL compiler directly. This is a basic tool provided by every ORB vendor.

In our account example, the developers of the account business object provide us with the IDL files `Account.idl` and `AccountKey.idl`.

The key, for a Java client developer, is to run the IDL files through the IDL-to-Java compiler.

We already are familiar with `Account.idl`, which describes our account business object, but not with `AccountKey.idl`, which is also generated by the Object Builder. For the moment, consider that we need to generate proxy classes for these two IDL files. For that purpose, CBConnector provides an IDL-to-Java compiler that is used from the command line:

```
java -classpath%CLASSPATH% COM.ibm.idl.toJava.Compile
-i%IDL_INCLUDE% drive:\CBConnector\Account.idl
```

The emitter produces Java source files and a Java package:

```
Account.java
AccountHelper.java
AccountHolder.java
_AccountStub.java
AccountPackage with
    NotEnough.java
    NotEnoughHelper.java
    NotEnoughHolder.java.
```

To generate the proxy classes for the `AccountKey.idl` file, we use the IDL-to-Java compiler with the following command:

```
java -classpath%CLASSPATH% COM.ibm.idl.toJava.Compile
-i%IDL_INCLUDE% drive:\CBConnector\AccountKey.idl
```

The emitter produces the following Java files:

```
AccountKey.java
AccountKeyHelper.java
AccountKeyHolder.java
```

Now we can start VisualAge for Java and import the source files into our VAJEnterpriseRedbook project. This operation results in the creation of a new package called `AccountPackage`, which contains the `NotEnough`, `NotEnoughHelper`, and `NotEnoughHolder` classes.

A default package is also created and contains the `_AccountStub`, `AccountHelper`, `AccountHolder`, `AccountKeyHelper`, and `AccountKeyHolder` classes, and the `Account` and `AccountKey` interfaces.

Before we continue, we are going to move all these classes and interfaces to a new package named `Account`.

Everything is not yet ready. You have probably noticed that the `AccountKeyHelper` class is marked by the Workbench with a red cross. After expanding the class, you see that the `_create` method has a problem. It returns an object of type `_AccountKeyImpl`, which does not exist. So we have to create it. Remember, after importing the Java files created by the IDL compiler, we got two

interfaces. The `Account` interface has an implementation called `_AccountStub`, but the `AccountKey` interface has no implementation. So let us create a new class that implements this interface.

We create a class named `_AccountKeyImpl`, with the superclass `COM.ibm.IManagedLocal_IPrimaryKeyImpl`, and we implement the `AccountKey` interface.

However, this is not all. We have extended `COM.ibm.IManagedLocal_IPrimaryKeyImpl`, and this abstract class requires implementing the `externalize_to_stream` and `internalize_from_stream` methods. We need these methods for when we invoke the `_toString` method that implicitly calls `externalize_to_stream` to generate a byte array that is shipped to the CBBConnector server. That is how a primary key is passed to the server object.

Here are the implementations:

```
public void externalize_to_stream (org.omg.CosStream.StreamIO arg1 ) {
    arg1.write_string(anAccount);
}

public void internalize_from_stream
    (org.omg.CosStream.StreamIO arg1,
     org.omg.CosLifecycle.FactoryFinder arg2)
    throws org.omg.CosStream.StreamDataFormatError {
    anAccount = arg1.read_string();
}
```

We also have to create the `String` variable called `anAccount`, which holds the account number used by the two methods. We do not describe this simple task! Now we implement the methods of the `AccountKey` interface:

```
public void accountNumber(String newAccountNumber) {
    anAccount=newAccountNumber;
}

public String accountNumber() {
    return anAccount;
}

public static _AccountKeyImpl _create ( ) {
    return new _AccountKeyImpl();
}
```


Creating the Java Client

The CBConnector account client is developed in two steps:

- ❑ Develop a nonvisual bean called BuildCBC, which encapsulates all CBConnector initialization up to the point where a factory able to create account instances is found.
- ❑ Develop an applet called AccountView allowing the user to create, update, and find account objects (Figure 195).

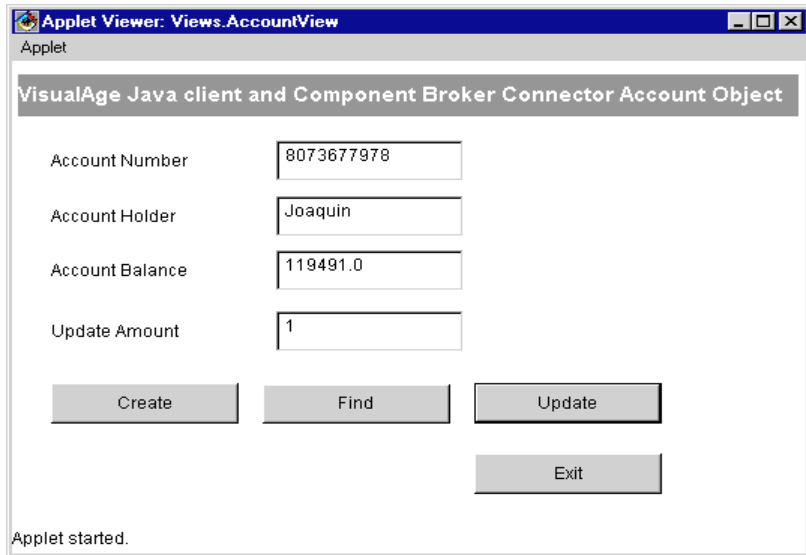


Figure 195. AccountView Applet

The user fills in the account number and the account holder and requests the creation of the server object by clicking on **Create**.

Once the server is created, the user can enter an update amount and click on **Update**. The account balance is updated from the previous balance and the update amount.

To retrieve an existing account, the user fills in the account number field and clicks on **Find**. In return, the account holder and the account balance are updated.

Error messages are displayed in three situations:

- ❑ If the balance becomes negative, the request is rejected, an exception is thrown, and a message box displays: AccountPackage.NotEnough.

- ❑ If an object already exists with the same account number, a message box informs the user that it is a duplicate key: `COM.ibm.ImanagedClient.IDuplicateKey`.
- ❑ If the specific account instance does not exist, a message box informs the user that no object exists with the specified key: `COM.ibm.ImanagedClient.INoObjectWKey`.

This trivial application example is sufficient to show how to develop a Java client that creates and uses objects on a CBCConnector server and to demonstrate the power of visual programming, a unique feature of the VisualAge family of tools!

Creating the Java Bean for the Applet

We create a new package named `Model` and a nonvisual bean that extends `Object` and call it `BuildCBC`. This bean is going to be passed two parameters required for the ORB initialization. Therefore, we create two properties that hold these parameter values:

- ❑ `orbArgs`, of type `java.lang.String[]`, holds the arguments provided in the command line (if started as an application) or the applet parameters.
- ❑ `orbProps`, of type `java.util.Properties`, holds the CBCConnector server host name and port number.

Now we define a method called `passingOrbArgs` to set `orgArgs` and `orbProps`:

```
public void passingOrbArgs(java.lang.String[] args,
                          java.util.Properties props) {
    setOrbArgs(args);
    setOrbProps(props);
}
```

Because `BuildCBC` encapsulates all of the initialization, we have to create an event that is used to trigger it. We create a new listener interface named `CBCinit`, with an `initializeCBC` method to be implemented.

VisualAge for Java generates most of the code to handle events. However, it cannot guess that we want to fire this event at the end of the `passingOrbArgs` method. We go back to the `passingOrbArgs` method and change the code to:

```
public void passingOrbArgs(java.lang.String[] args,
                          java.util.Properties props) {
    setOrbArgs(args);
    setOrbProps(props);
    fireInitializeCBC( new CBCinitEvent(this) );
}
```

Now, we open the Visual Composition Editor for the BuildCBC bean. From an empty free-form surface, we build step by step what is shown in Figure 196. The flow is from top left, down, and then to the right.

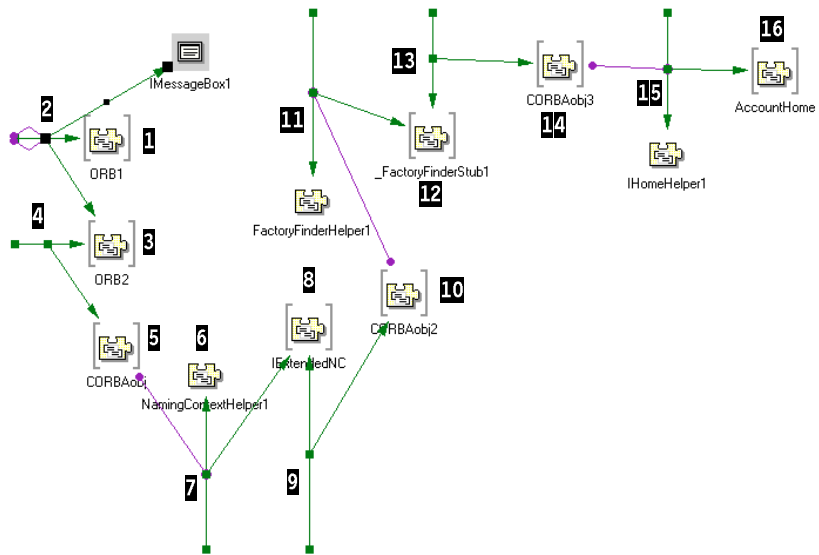


Figure 196. Visual Composition of BuildCBC

We start by initializing the CBConnector environment; that is, getting access to the Java ORB:

- ❑ We add a bean variable of type `COM.ibm.CORBA.iiop.ORB` (ORB1) (1)
- ❑ We connect the `CBCinit` event of the free-form surface to the `init(java.lang.String[],java.util.Properties)` method of ORB1. The two parameters are the `orbArgs` and `orbProps` properties of the free-form surface (the BuildCBC bean) (2).
- ❑ We drop a second `COM.ibm.CORBA.iiop.ORB` variable (ORB2) and connect the `normalResult` of the `init` method to its `this` property. In fact we use a static method of the ORB class to create a new instance returned in ORB2 (3).

Now we have a Java ORB in our client machine and a connection set up during the initialization with what is called a Bootstrap server. The Bootstrap server becomes our anchor point.

As you already know, CORBA provides naming services used to create well-known objects registered in the naming tree. So, we access this naming tree and retrieve a remote object reference to the name space root. We hold this object reference in a variable called `iExtendedNC`.

- ❑ We connect the CBCinit event of the free-form surface to the `resolve_initial_references(java.lang.String)` method of ORB, and we set the parameter to "NameService." (4)
- ❑ We drop a variable of type `org.omg.CORBA.Object` (CORBAObj.) and connect the `normalResult` of the previous connection to CORBAObj. (5)
- ❑ We add a bean of type `COM.ibm.IExtendedNaming` (NamingContextHelper1) (6).
- ❑ We connect the CBCinit event of the free-form surface to the `narrow` method of NamingContextHelper1 and connect CORBAObj to the parameter (7).
- ❑ We add a variable of type `COM.ibm.IExtendedNaming.NamingContext` (IExtendedNC) and connect the `normalResult` of the previous connection to IExtendedNC (8).

Now IExtendedNC gives us access to objects registered in the naming tree. During the installation of CBCConnector, some well-known objects are bound in the naming tree. One of these objects is a default FactoryFinder.

A factory finder knows how to retrieve factories that can create instances for specific interfaces (IDL). The default factory finder is identified in the naming tree by "host/resources/factory-finders/host-scope." This is what we could use if we had only one factory that could create instances for the account interface. However, in our case, the Account interface has been associated with its own factory finder. We do not use the default factory finder but instead "host/resources/factory-finders/AccountTRScope."

- ❑ We connect the CBCinit event of the free-form surface to the `resolve_with_string(java.lang.String)` method of IExtendedNC and set the parameter to "host/resources/factory-finders/AccountTRScope" (9).
- ❑ We drop a CORBA object variable, call it CORBAObj2, and connect the `normalResult` to CORBAObj2 (10).
- ❑ We drop a `COM.ibm.IExtendedLifeCycle.FactoryFinderHelper` bean (FactoryFinderHelper1), connect the CBCinit event to the `narrow` method, and pass CORBAObj2 as a parameter (11).
- ❑ We add a `COM.ibm.IExtendedLifeCycle._FactoryFinderStub` variable (_FactoryFinderStub1) and connect the `normalResult` to _FactoryFinderStub1 (12).

What we are looking for is a factory that can create instances of the account class.

Every CBConnector managed object class has an instance of a factory associated with it. The factory provides a set of interfaces for creating instances of a managed object. The factory gets some of its interface from the base class, `CosLifeCycle.GenericFactory`. The method we will use (`createFromPrimaryKeyString`) is part of the `IManagedClient.IHome` interface supplied by CBConnector. This interface specializes the `COSLifeCycle.GenericFactory` interface and plays the role of a factory for CBConnector-managed objects. Object providers can implement and provide a tailored subclass of this interface, or they may simply use the implementation of `IHome`. A client programmer must know how to find the right `IHome` for object creation. Homes are at well-known locations in the name service. The input required for the factory finder is the name of the implementation class for which we want this factory to create instances.

There are several ways to retrieve the home account factory. We are going to use `find_factory_from_string` and pass a parameter specifying the IDL interface in which we are interested.

- ❑ We connect the `CBCinit` event of the free-form surface to the `find_factory_from_string` method of the `_FactoryFinderStub1` and set the parameter to "Account.object interface" (13).
- ❑ We drop a CORBA object variable (`CORBAobj3`), and connect the `normalResult` to `CORBAobj3` (14).
- ❑ We drop a bean of type `COM.ibm.IManagedClient.IHomeHelper` (`IHomeHelper1`), connect the `CBCinit` event to its `narrow` method, and pass `CORBAobj3` as a parameter (15).
- ❑ We add a variable of type `COM.ibm.IManagedClient.IHome` (`AccountHome`) that will hold the searched account home and connect the `normalResult` to `AccountHome` (16).

At this point the `BuildCBC` bean is finished. `AccountHome` is the factory we need to create account instances on the server. We have to promote two methods of `AccountHome` so that they can be used by the GUI: `findByPrimaryKeyString` and `CreateFromPrimaryKeyString`.

Creating the Applet

We now have the BuildCBC bean, which encapsulates all of the initialization and the lookup for an account factory.

It is time to create a GUI in the form of an applet. We create a Views package and an applet named AccountView (Figure 197).

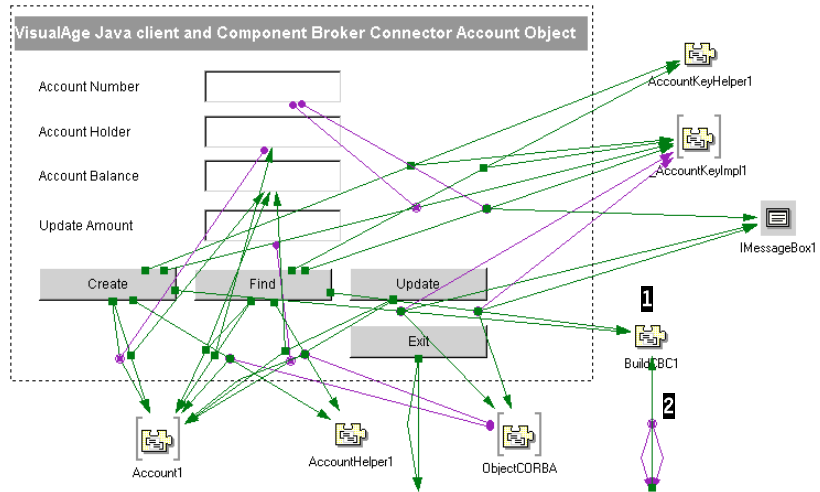


Figure 197. Visual Composition of the AccountView Applet

We start by creating two methods, `buildArgs` and `buildProps`, that will provide the required information for the ORB initialization:

```
public java.lang.String[] buildArgs() {
    String[] theArgs = new String[2];
    theArgs[0] = getParameter("hostName");
    theArgs[1] = getParameter("port");
    return theArgs;
}

public java.util.Properties buildProps() {
    /* Perform the buildProps method. */
    java.util.Properties props = new java.util.Properties();
    String serverHostName = "";
    String serverPort = "";
    serverHostName = getParameter("hostName");
    serverPort = getParameter("port");
    props.put("org.omg.CORBA.ORBClass", "COM.ibm.CORBA.iiop.ORB");
    props.put("com.ibm.CORBA.BootstrapHost", serverHostName);
    props.put("com.ibm.CORBA.BootstrapPort", serverPort);
    return props;
}
```

We use the Visual Composition Editor to build the applet:

- ❑ We drop a BuildCBC bean (BuildCBC1) (1).
- ❑ We connect the applet's init event to the passingOrbArgs method of BuildCBC1 and connect the two parameters to the applet's buildArgs and buildProps methods (2).

We chose to use the init event generated by the browser to call the passingOrbArgs method, which in turn generates the CBCinit event used by most of the beans in BuildCBC.

When we return from this method, the ORB is initialized, we have access to the naming tree, and we have access to the account factory called AccountHome.

Creating Account Objects

Before going into the process of manipulating account objects, let us build the GUI interface shown in Figure 198.

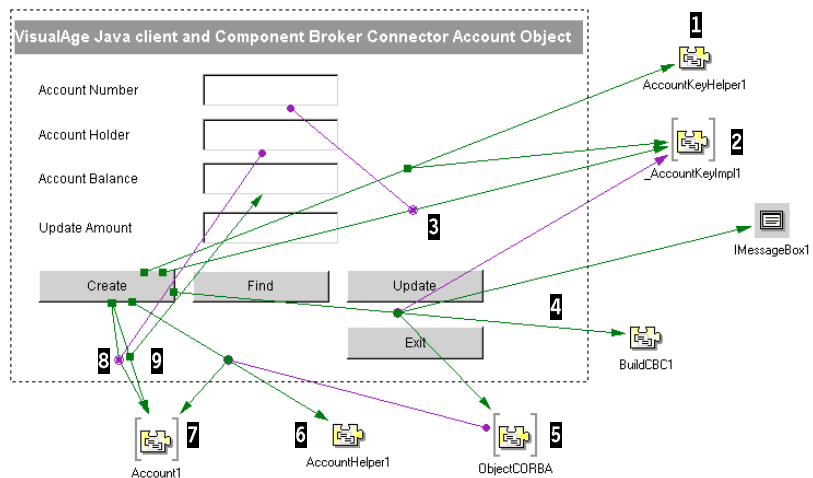


Figure 198. Creation of Account Objects

We drop four labels, entry fields, and push buttons.

CBConnector managed objects can be created in a number of ways. In the sections that follow we describe the default way in which we can easily create managed objects.

Before we can create an account object, we need to prepare a key. In general, only a very small subset of the object instances in a distributed system will be in the name service. These will typically be large well-known objects such as collections of business objects or important object instances in the object model.

The client programmer creates a primary key object to define the identity of the object that will be created. Then, the `createFromPrimaryString` call is made on the home and the key is passed. The `createFromPrimaryString` method is defined by the `IHome` class, and all business objects can be created by this method. Instances of a given class or home are managed by the home through primary keys.

Key helper classes are generated by using the IDL-to-Java emitter run on IDL files. In this case, the key helper class is named `AccountKeyHelper` and has been generated from `AccountKey.idl` provided by the implementer of the account server object.

In our example, the creation of an account object is done by using a primary key. This key is created by using an `AccountKeyHelper` and you probably remember that earlier we provided an implementation for this class generated by the IDL-to-Java compiler.

All user-defined IDL types have a helper Java class with the suffix `Helper` appended to the type name generated. Several static methods needed to manipulate the type are supplied. These include any insert and extract operations for the type, getting the repository ID, getting the type code, narrowing, and reading and writing the type from and to a stream.

All key instances are created with a static method call named `_create`. Once an instance of a primary key is created, the key must be set by one or more attributes on the primary key object. Once all of the key attributes have been set, the primary key object is now usable. The account home uses this primary key to create the account object.

- ❑ We drop an `AccountKeyHelper` bean and connect the **Create** button to the `_create` method (1).
- ❑ We drop an `_AccountKeyImpl` variable and connect the normalResult to `_AccountKeyImpl`. After calling the static class method `_create`, we have an `_AccountKeyImpl` instance that is used to set the account number (2).
- ❑ We connect the **Create** button to `_AccountKeyImpl AccountNumber(java.lang.String)` and pass the account number entry field as a parameter (3).

At last, we are ready to create an instance of account in the server. We call `createFromPrimaryString` on the `AccountHome` object. Remember this method has been promoted from `AccountHome`, which is itself encapsulated in `BuildCBC`.

- ❑ We connect the **Create** button to the `createFromPrimaryString` method of `BuildCBC`. This method returns a CORBA object that represents an account (4).

- ❑ We drop a CORBA object variable (ObjectCORBA) and connect the normalResult to ObjectCORBA (5).

To ensure that the returned object is really an account, we need to use an AccountHelper. This class has a narrow method that checks the type and returns a proxy of the requested object.

- ❑ We add an AccountHelper bean (AccountHelper1), connect the **Create** button to the narrow method, and pass the ObjectCORBA as a parameter (6).
- ❑ We add an Account variable (Account1) and connect the normalResult to Account (7).

We are done! The result of the narrowing is a proxy of the account object just created on the server.

The first thing we do with the account is to set the account holder and the balance:

- ❑ We connect the **Create** button to the AccountHolder method and get the parameter from the account holder entry field (8).
- ❑ We connect the **Create** button to the getBalance method and connect the normalResult to the balance entry field (9).

Updating Account Objects

In this section we show you how to update the balance of an account object.

Figure 199 shows the connections for the update function.

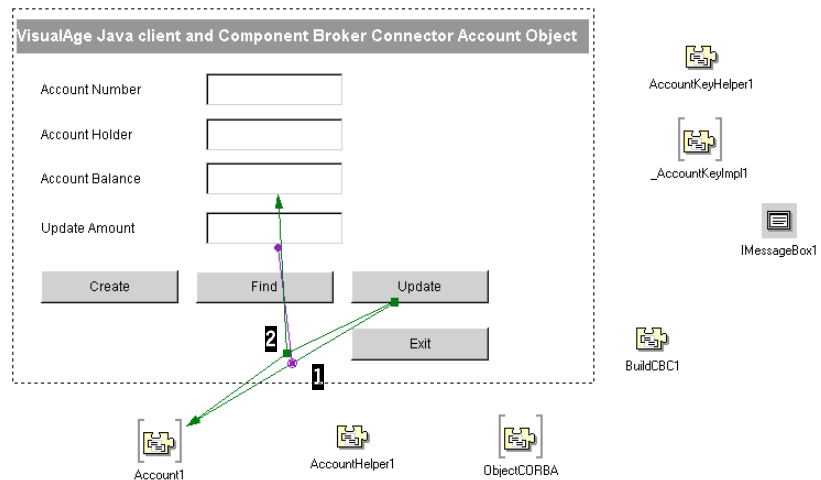


Figure 199. Updating Account Objects

For this purpose:

- ❑ We connect the **Update** button to the `changeBalance` method and pass the amount field as a parameter (1).
- ❑ We connect the **Update** button to the `getBalance` method and connect the `normalResult` to the account balance field (2).

Finding Account Objects

In this section we add the connections so that we can find account objects based on the account number. Once objects have been created from any client, we might want to retrieve them (Figure 200).

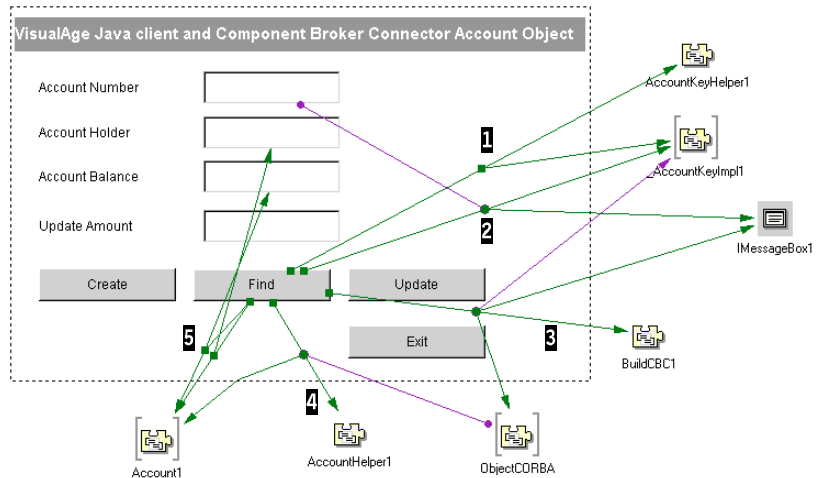


Figure 200. Finding Account Objects

Because the first user action, after the applet is started, may be to find an object, we need to create an instance of `_AccountKeyImpl`:

- ❑ We connect the **Find** button to the `_create` method of `AccountKeyHelper` and connect the `normalResult` to `_AccountKeyImpl1` (1).
- ❑ We set the account number from the corresponding text field by connecting the **Find** button to the `accountNumber` method of `_AccountKeyImpl` (2).
- ❑ To find the object, we invoke `findByPrimaryString` on the `BuildCBC` bean. We connect the **Find** button to the `findByPrimaryString` method of `BuildCBC`, passing `AccountKeyImpl.toString` as a parameter, and we connect the `normalResult` to `ObjectCORBA` (3).
- ❑ We narrow the result by connecting the **Find** button to the `narrow` method of `AccountHelper1`, passing `ObjectCORBA` as a parameter, and we connect the `normalResult` to `Account1` (4).

- ❑ We update the account holder and balance text fields with two connections in the same way as in “Updating Account Objects” on page 349 (5).

Releasing and Deleting Objects

Eventually, a client application will no longer need an object that was created or found. CBBConnector supports two ways of cleanup:

- ❑ The remove method asks the object to delete itself and its instance data.
- ❑ The release method informs the object that the client application no longer references the object. The object still exists in the server, and other applications may use it.

This is easy enough so we do not explain how to implement the cleanup.

Now we can run the applet and start playing with CBBConnector objects running in a distributed environment!

12

Deployment of Java Applications and Applets

In this chapter we describe the steps that are necessary to deploy applications and applets from VisualAge for Java.

Deployment of Applications

The basic characteristics of applications are:

- ☐ They run in a platform JVM.
- ☐ They must be installed on a client or server machine.
- ☐ They have normal access to the machine they run on and to other machines on the network.

Prerequisites for Applications

To run applications deployed from VisualAge for Java, the machine must have:

- ☐ A JVM; VisualAge for Java requires at least JDK 1.1.1
- ☐ VisualAge for Java run-time libraries if Enterprise Access Builders were used. Copy the file to the machine and add it to the CLASSPATH environment variable:

```
d:\IBMJava\eab\runtime\ivjeab.zip
```

Design for Portability

If an application has to run on multiple platforms, you must observe certain restrictions:

- ☐ No native methods, that is, no C++ or other non-Java language calls
- ☐ No calls to the operating system
- ☐ Use Java support for portability, for example, to retrieve the separator character (forward or backward slash) for file names code:

```
String sep = System.getProperty("file.separator")
```

Exporting an Application from VisualAge for Java

The steps to export an application are:

- ☐ Create a master directory for the exported classes.
- ☐ Select the classes in the Workbench and export the class files. Use the *Create package subdirectories* option so that classes of the same package are exported into the same subdirectory of the master directory.

Deployment Process for Applications

Copy the exported directories to the machine where the applications will run. Set up a master directory that is in CLASSPATH and copy the package subdirectories into the master directory.

Figure 201 shows the process of deploying applications from a VisualAge for Java development machine to an execution machine.

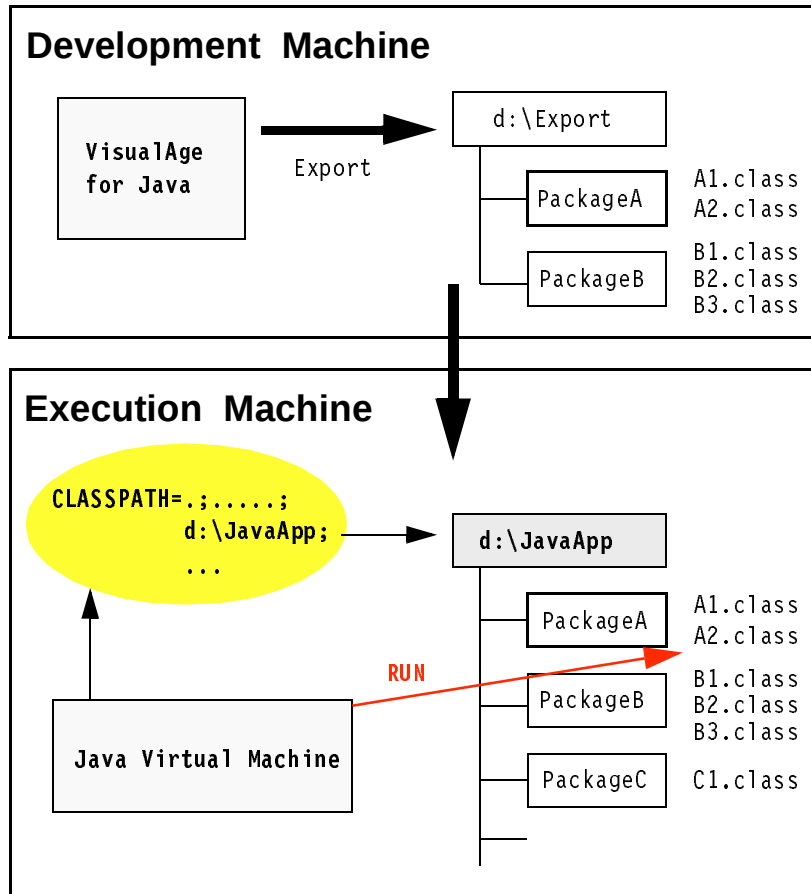


Figure 201. Application Deployment Process

A setup like that in Figure 201 guarantees that all classes are found by the JVM. A proper setup that follows the Java class naming rules is required for successful operation.

Deployment of Applets

The basic characteristics of applets are:

- ☐ They run in a Web browser on a client machine.
- ☐ They are downloaded from a server machine.
- ☐ They have limited access to the client machine.
- ☐ They have limited access to the network—only to the server machine from which they come.

Note: Some of the restrictions can be lifted for trusted applets; such security facilities are, however, beyond this document.

The biggest advantage of applets is that they are automatically distributed to the client machine. There is no maintenance burden; new versions of applets have to be installed only on Web servers.

Exporting an Applet from VisualAge for Java

The steps to export an applet are the same as for an application:

- ☐ Create a master directory for the exported classes.
- ☐ Select the classes in the Workbench and export the class files. Use the *Create package subdirectories* option so that classes of the same package are exported into the same subdirectory of the master directory.

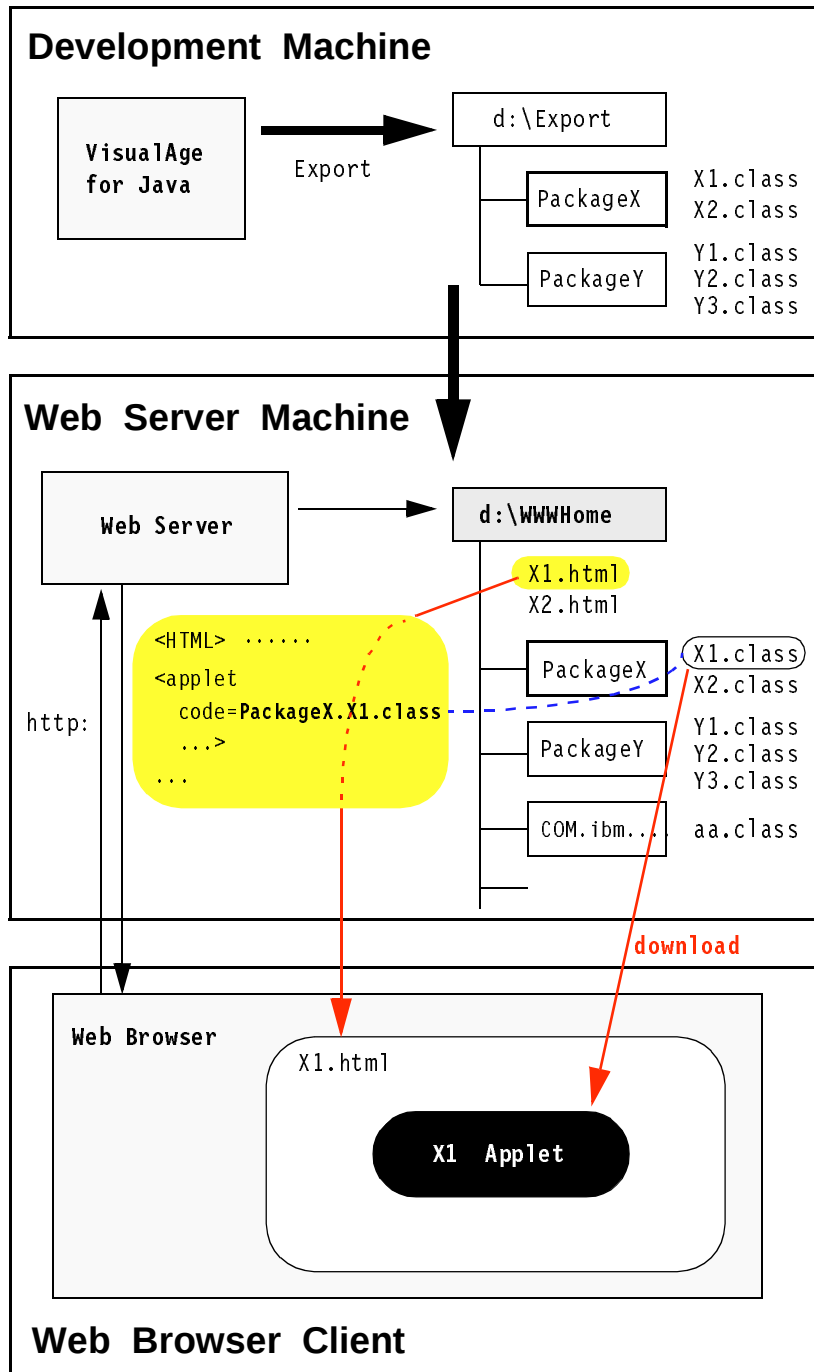
Deployment Process for Applets

Copy the exported directories to the Web server machine. Set up a master directory where the Web server looks for HTML files and applets and copy the package subdirectories into the master directory.

Expand the VisualAge for Java enterprise library and the DB2 JDBC library into a subdirectory called COM\ibm of the master directory. See “Run-time Libraries” on page 358 for detailed instructions.

In the master directory, create an HTML file for each applet that points to the applet's class.

Figure 202 shows the process of deploying applets from a VisualAge for Java development machine to a Web server and a Web browser.

**Figure 202.** Applet Deployment Process

Run-time Libraries

For applets that use Enterprise Access Builder functions or JDBC, the Web server needs access to the run-time libraries so that the run-time classes can be downloaded with the applet.

When you expand the enterprise access library (ivjeab.zip) and the DB2 JDBC library (db2java.zip) into the master directory of the Web server, you end up with this directory structure:

```
COM\ibm\ivj\eab\cics
                  \data
                  \j2cpp
                  \rmi\client
                  \javad
                  \server
                  \javabeans

COM\ibm\db2\app
               \jdbc\app
               \net
```

Jar Files

As an alternative to downloading individual runtime classes with the applets, you can deploy the following compressed Java archive files:

- ❑ ivjdata.jar—contains the core set of data access classes, without any editors, forms, access application, or BeanInfo classes (COM.ibm.ivj.eab.data)
- ❑ ivjbeans.jar—contains the javabeans classes (COM.ibm.ivj.javabeans)

The names of these archive files can be added in the HTML code that accesses the applet. The classes that are in the files will be accessed faster than they would be if the applet was looking for them on the server. Classes that are not in the archive files are still accessed on the server. The disadvantage is that potentially too many classes are downloaded with the archive; however, the browser caches them for other enterprise applets.

A

Installation, Setup, and Prerequisites

In this appendix we describe the prerequisites for JDBC applications and the installation and setup activities for CICS components and redbook samples.

Installation of VisualAge for Java

We do not cover the basic product installation in this book because it is straightforward.

Prerequisites for JDBC Applications

In this section we present the prerequisites for developing and running JDBC applications.

DB2 Prerequisites

The prerequisites to develop and run JDBC applications with DB2 are:

- ❑ DB2 must be installed on a server, or on a stand-alone client. DB2 must be at Version 2.1.2 with a fix pack, or DB2 Universal Database (UDB).
- ❑ In a two-tier architecture, you can have a DB2 client on the client machine, and a DB2 server on a server; or you can have the client with just a browser, and the DB2 client and server on the server.
- ❑ In a three-tier architecture, you have a client with a browser, a middle tier with a DB2 client, and a DB2 server.
- ❑ Make sure DB2 is started. Issue DB2START on OS/2 and Windows 95. On Windows NT issue NET START DB2 and NET START DB2NTSECSEVER (or have DB2 automatically started as a service).
- ❑ Start the DB2 Java daemon on the DB2 client machine. Issue the DB2JSTRT 8888 command, where 8888 is the port we use for DB2 JDBC communication.
- ❑ Make sure that the db2java.zip file is in CLASSPATH. This file is located in d:\SQLLIB\JAVA, where SQLLIB is the DB2 installation directory.

VisualAge for Java Prerequisites

To develop and run JDBC applications with VisualAge for Java and DB2, you must import the DB2 Java classes into the Workbench:

- ❑ Expand the db2java.zip file (in d:\SQLLIB\JAVA) into subdirectories. Use the -d option in most unzip utilities. This creates a subdirectory structure, COM\ibm\db2\jdbc\...
- ❑ Create a project in the Workbench, for example, IBM DB2 JDBC Drivers.
- ❑ Select the new project and import the COM directory with all subdirectories.

ODBC Prerequisites

To use ODBC in your JDBC applications, you have to install ODBC drivers when you install VisualAge for Java. You also have to use the ODBC Data Source Administrator (available from the Windows control panel) and its associated online help to configure your data sources. Note that VisualAge for Java automatically does this for you in the case of the SAMPLEDBASE database that is shipped with the product.

You have to specify an ODBC driver (for example, `sun.jdbc.odbc.JdbcOdbcDriver`) in the Access page of the Data Access Builder's Property pop-up menu for the database bean.

For more information about configuring ODBC drivers (including considerations for Sybase and Oracle) see the online help for VisualAge for Java. Select *Tasks -> Accessing Enterprise Data -> Relational Databases -> 6. End a Data Access Builder Session* and look for the Deploy Applications topic.

VisualAge for Java Enterprise ships Intersolv ODBC drivers with the product.

Installation and Setup of CICS Components

The setup for a VisualAge for Java and CICS configuration includes three CICS components:

- ☐ CICS server
- ☐ CICS Client
- ☐ CICS Java Gateway

We do not discuss the installation of a CICS server, assuming that it already exists.

There are several sources for the CICS Client and CICS Java Gateway products, including CD-ROM. We chose to download them over the Web:

<http://www.hursley.ibm.com/cics/downloads/index.html>

This Web site offers the latest versions, provided that you are a licensed user of appropriate software. The download registration process gives you all the details. You can download the *CICS Gateway for Java* as a separate item, or the *CICS Client for Windows NT*, which includes the Gateway.

We downloaded the *CICS Client for Windows NT*, Version 2.0.2, update UQ07699. You need to download LOADDSKF.EXE from the same source and use it to load each of the five downloaded files to diskette, for example:

```
loaddskf c:\download\cc Clint_1.exe a:
```

Installing the CICS Client for Windows NT

You install the CICS Client on Windows NT by running `a:INSTALL`. Select an execution directory, for example, `C:\CICSCLI`.

We need a CICS server, of course, for our CICS client to talk to. The definitions below cover two servers:

- ☐ An enterprise host CICS/ESA server
- ☐ A workstation CICS for OS/2 server

The CICS for OS/2 server is not officially supported as a target for the CICS Access Builder in VisualAge for Java 1.0. We supply it here as an example of an easily configured test system that communicates through TCP/IP. We found it useful, provided we transferred only character-coded data, for testing our applications with a CICS for OS/2 server, instead of splitting our

attention across a CICS Gateway for Java, a CICS Client for Windows NT, and an APPC (LU 6.2) communication link to a CICS/ESA host.

Configuring the CICS Client for TCP/IP Connections

You need to configure the client. Our recommendation is to modify a copy of C:\CICSCLI\BIN\CICSCLI.INI and set the CICSCLI environment variable to point to this copy. You have to enter the host name or IP address of your CICS server (or servers) and give your client a name that will be used for the terminal resource created in the CICS server by the AutoInstall process.

You specify a value for *MAXBUFFERSIZE*, the size of a communication buffer (in KB); in particular, it limits the size of the COMMAREA your application programs can pass through the CICS Gateway for Java. The valid range is 4 KB to 32 KB, and the default is 32 KB. Therefore, **the maximum amount of data you can pass from Java over the CICS Gateway for Java to the CICS application is 32 KB in each call.** (This is a CICS restriction on the size of COMMAREAs, rather than a Gateway limitation.)

In our configuration, the CICS server has the TCP/IP host name *banda.almaden.ibm.com*. Our Web server machine, which carries the CICS Client for Windows NT and the CICS Gateway for Java, has the host name *senegal.almaden.ibm.com*.

Our copy of CICSCLI.INI is called SENEGAL.INI, and it reads:

```

;* IBM CICS Client - Initialization File
Client = SENEGAL           ; Auto-install client on the server
MaxServers = 1             ; Only allow one server connection
MaxRequests = 20           ; Limit maximum server interaction
MaxBufferSize = 32        ; Allow for a 32K maximum COMMAREA
LogFile = CICSCLI.LOG      ; Set the error log file name
TraceFile = CICSCLI.TRC    ; Set the trace log file name
DumpFile = CICSCLI.DMP     ; Set the memory dump file name
DumpMemSize = 16          ; Allow for 16k of trace in memory

Server = BANDA             ; Arbitrary name for the server
Description = Banda CICS TCP/IP Server; Arbitrary description
Protocol = TCPIP           ; Matches a Driver section below
NetName = banda.almaden.ibm.com ; The server's TCP/IP address
Port = 0                   ; Use the default TCP/IP CICS port

Driver = TCPIP             ; Matches the Server's Protocol value
DriverName = CCLWNTIP      ; Use WinNT TCP/IP comm.s DLL
; End of initialization file

```

We can test that the client works in three steps:

- ❑ Run `CICSCLI /s`, which starts the client.
- ❑ Run `CICSCLI /s=banda`, which starts the connection to the banda CICS server.
- ❑ Run `CICSCLI /l`, which lists the active server status; it should show the banda machine active.

The installation program registers the client as a Windows NT service. After rebooting the server, we can configure it to automatically start the CICS client.

Configuring the CICS Client for APPC Connections

If you are working with an Enterprise CICS system, you are probably working with CICS/ESA on a System/390 architecture host. CICS/ESA supports Systems Network Architecture—Advanced Program to Program Communications (SNA APPC) links instead of TCP/IP for the CICS Client for NT. (You can also use a hybrid, TCP62, which looks like TCP/IP at the CICS Client end and APPC at the enterprise server end.) The setup details are very different, though once set up, you test it exactly as before.

The CICS Client for NT requires another layer of software to provide the SNA communications. It supports:

- ❑ IBM Communications Server Version 5.0 or later, which may be installed on the workstation; or Communications Server Client for Windows NT, which may be installed on the CICS Client for Windows NT workstation.
- ❑ IBM Personal Communications for Windows NT Version 4.2 (or Version 4.1 with an APAR fix).
- ❑ Microsoft SNA Server Version 2.11, or later, with Service Pack 1 or later, which may be installed on the workstation; or SNA Server Client, which may be installed on the CICS Client for Windows NT workstation.

We used IBM Personal Communications Version 4.2. The definitions across it, the CICS Client, the CICS/ESA system and SNA VTAM must be synchronized. The numbers in the tables that follow identify parameters that should have the same values.

The five programs involved have different configuration tools:

- ❑ The VTAM network definitions at the CICS/ESA system are defined in library members in the VTAMLST file; in particular, the application definitions in the ATCSTRxx member, where xx is usually 01 (Table 21).

- ❑ Personal Communications has a node configuration program that uses Windows notebook pages for each parameter (Table 22).
- ❑ CICS/ESA provides the CEDA transaction which has a line-mode command language and paged panel display for almost all CICS system resources (Table 23).
- ❑ The CICS Client for NT parameters are defined in an .INI file (Table 24).
- ❑ The VisualAge for Java application parameters are defined in the CICS Logical Unit of Work bean (Table 25).

Table 21. VTAM Definitions for the SNA Network in CICS/ESA

ID	Parameter	Example
1	NetID: the name of the SNA domain containing the CICS/ESA system. This is in the ATCSTRxx member of the VTAMLST library used to start the SNA VTAM node.	GBIBMFG
2	APPL: the CICS system's logical unit name	FGBZ1C32
3	LU: the CICS client's logical unit name for the LU 6.2 connection	SC02JAVA
4	LogMode: the logmode entry suitable for LU 6.2. LU62PS is usually correct.	LU62PS

Table 22. Personal Communications Definitions

ID	Parameter	Example
1 2	Partner LU 6.2: the full name (two components separated by a dot, "domain.LU") for the server CICS/ESA system	GBIBMFG FGBZ1C32
3	Local LU 6.2: the CICS client's LU name. Leave the <i>dependent</i> flag unset.	SC02JAVA
4	Mode: the logmode for LU 6.2	LU62PS

Table 23. CICS/ESA Definitions

ID	Parameter	Example
2	APPLID: set in the System Initialization Table (SIT)	FGBZ1C32
5	Connection name: an arbitrary name	JAVA
3	Connection netname: the CICS client's logical unit name for the LU 6.2 connection. Use CEDA to define it.	SC02JAVA
5	Sessions connection: refers to the connection for which this session specifies parameters. Use CEDA.	JAVA
4	Sessions modename: the logmode entry suitable for LU 6.2. LU62PS is normally correct. Use CEDA.	LU62PS

Table 24. CICS Client INI File Definitions

ID	Parameter	Example
1 . 2	NetName: the full name (two components separated by a dot, "domain.LU") for the server CICS/ESA system	GBIBMFG . FGBZ1C32
3	Local LUName: the CICS client's logical unit name for the LU 6.2 connection	SC02JAVA
5	Protocol: an arbitrary name referring to the appropriate Driver statement in the .INI file	SNA
4	ModeName: the logmode entry suitable for LU 6.2. LU62PS is usually correct.	LU62PS
5	Driver: agrees with the Protocol name	SNA
6	DriverName: CCLWNTSN is the APPC driver for Windows systems.	CCLWNTSN
7	Server: an arbitrary name for the CICS server, used in the CICSCLI /S= start command	CICSHost

Table 25. VisualAge for Java CICS Unit of Work Bean Properties

ID	Parameter	Example
7	dest: the server name for the CICS host system in the CICS Gateway for Java .INI file	CICSHost
8	gatewayHostName: the TCP/IP name for the system running the CICS Gateway for Java	thulium. almaden. ibm.com
9	targetCodePage: the host CICS system's code page. Only page IBM-037 is supported in VisualAge Version 1.0.	037
10	Userid: the user ID, registered on the host CICS system, under which the host transaction will run	USERID
11	Password: password for the host CICS system	PASS WORD

Installing the CICS Gateway for Java

The Gateway code comes as a single self-extracting zip file, JG-11NT.EXE, in the CICSCLI\JAVA directory. Running it creates a deep directory structure starting at CICSCLI\JAVA\JGate\... You can move the JGate directory to the boot drive to slightly simplify things:

```
move CICSCLI\JAVA\JGate c:\
```

The installation and configuration information is in the User's Guide, an HTML file. You have to add JGate\classes to the CLASSPATH environment variable.

We start the gateway by running Jgate.cmd from the JGate\bin directory, and we test the gateway to see whether it finds the CICS server:

```
java ibm.cics.jgate.test.TestECI jgate=senegal status
```

Installation of the Redbook Samples

The samples used in this document are distributed in a zip file called sg245081.zip. Unzipping the file on a hard drive creates an SG245081 directory with subdirectories as listed in Table 26.

Table 26. Redbook Sample Code

Directory	Subdirectory	Description
DAT		Import files for VisualAge for Java (see Table 27 for details)
JDBC		Supporting material for JDBC
	AtmDB	Defines the DB2 ATM database and tables and load the sample data
	JdbcConnect	JDBC connection sample
	SampleOrg	Sample ORG application (native and with Data Access Builder)
	OrgApplet	Sample ORG applet
	JdbcTest	JDBC update, insert, delete sample
	StatementTest	Prepared statement
	SampleApplication	ORG applet exported from VisualAge for Java
	SampleDax	Beans generated by Data Access Builder
	StoredProc	Stored procedure example
RMI		Supporting material for RMI
	RMINative	RMI native sample
	RmiPrimer	RMI account sample
	Scrapbook	Java code for testing the model
CICS		Supporting material for the CICS
	Cicscli	Sample .INI files for clients
	Cobol	COBOL sample source
	Pli	PL/I sample source
CPP		Supporting material for JNI/C++
	NativeTest	Sample using Java native interface
	NativeOps	Hypotenuse sample
	Reverse	Reverse string sample
	IString	Samples using IString
	CPPCard	ATM sample with C++ server

The import files can be loaded into VisualAge for Java. We used a project named VAJEnterpriseRedbook. The samples are structured into the files and packages listed in Table 27.

Table 27. Packages of the Redbook Samples

File	Package	Description
jdbc.dat	SampleDax	Data Access Builder classes for JDBC samples
	SampleApplication	JDBC sample applet and application
	ATMDax	Data Access Builder classes for JDBC ATM application
	ATMApplication	GUI classes for JDBC ATM application
rmi.dat	RMINative	RMI native sample
	RmiPrimer	RMI account sample
	RmiDax	Data Access Builder classes for RMI ATM application
	RmiModel	Business model classes for RMI ATM application
	RmiGui	GUI classes for nondistributed ATM application
	RmiGuiD	GUI classes for distributed ATM application
cics.dat	AtmCICSAccess	CICS Access Builder ATM application sample
cpp.dat	cppserver	C++ Access Builder reverse string sample
	stringtest	C++ Access Builder samples using IString
	cardserver	C++ Access Builder ATM application sample
cbc.dat	Account	CBConnector account beans
	AccountPackage	CBConnector generated classes
	Model	BuildCBC beans and events
	Views	CBConnector AccountView applet

A large, white, serif capital letter 'B' is centered within a gray square. The square is positioned on the left side of the page, and the letter 'B' is the first letter of the title 'Enterprise Access Builder'.

Enterprise Access Builder Changes in Version 1.0.1

This book was developed with the VisualAge for Java Enterprise Version 1.0 generally available since July 1997.

Version 1.0.1 became available January 1998. As the main feature Version 1.0.1 provides support for national languages, but it also includes enhancements for the enterprise access builders.

In this appendix we summarize the enterprise access builders improvements in Version 1.0.1. You can find the complete description in the release notes of the product.

AS/400 Feature

VisualAge for Java Enterprise 1.0.1 includes a unique AS/400 Feature that supports Java development for both client and server applications. The AS/400 Feature runs on Windows 95 and Windows NT and enables you to:

- ❑ Create a Java GUI for existing 5250 displays

The 5250 to AWT conversion SmartGuide converts your existing Data Description Specification (DDS) display files of your current RPG or COBOL program into Java AWT files. In just a few minutes you can get your first Java GUI screens without writing a single line of code. This will jump start your Java project and facilitate the move. This function requires Application Development ToolSet/400 (ADTS/400) V3R2 or later on the AS/400 machine.

- ❑ Call your AS/400 programs in Java programs

The Remote Program Call SmartGuide enables you to reuse your hundreds and thousands of lines of RPG and COBOL code with no additional coding. Simply fill in the AS/400 name, the program your Java program wants to call, and the parameters you want to pass. The SmartGuide generates the code and performs the data conversion between AS/400 data type and Java data type as well.

- ❑ Deploy Java programs to AS/400

Using the Export and Compile SmartGuides, Java files can be exported to AS/400 IFS and compiled to machine instructions for better performance. When you run your Java application on AS/400, you can use the cooperative debugger to debug the AS/400 Java program on the workstation. It is the same powerful, graphical debugger found in CODE/400.

- ❑ Immediately use the IBM Toolbox for Java classes, which are available in VisualAge for Java

The AS/400 Toolbox for Java is a set of Java programs that enables the Internet programming model. These programs can be used to easily access AS/400 data and resources, such as DB2/400 data, data queues, the integrated file systems, RPG or COBOL programs, batch commands, and print queues.

All Toolbox for Java classes are loaded into the VisualAge for Java Workbench at install time. Thus you can use the classes in the workbench as well as in the Visual Composition Editor without downloading from the AS/400 and importing into VisualAge for Java.

This information was extracted from:

<http://www.software.ibm.com/ad/as400/vajava>

Data Access Builder

A few changes affect the user-defined methods window of the Data Access Builder.

The windows previously used to add and modify user-defined methods have been changed to notebooks to improve readability. The same fields exist in the notebook as in Release 1.0, but now they are on three or four pages of the notebook, depending on the type of user-defined method.

There is a new Display Name field to specify a name for the method. This field can contain double-byte characters.

If you want to add a comment to the new user-defined method, you can enter it on the Comment Page of the notebook (rather than by clicking on the Comment icon).

There are notebooks for building:

- ☐ Customized SQL statements
- ☐ SQL predicate methods
- ☐ Stored procedure call methods

CICS Access Builder

A number of improvements have been made to the CICS Access Builder.

Changes to the IVJCicsUOWInterface Class

The CICS unit of work bean has been improved in several ways.

Additional Version of Transaction Invocation

There are additional versions of the `invokeTxn` and `asynchInvokeTxn` methods with the following signatures:

```
invokeTxn(IVJCicsEciCommArea In, IVJCicsEciCommArea Out)
asynchInvokeTxn(IVJCicsEciCommArea In, IVJCicsEciCommArea Out)
```

The new methods provide support for transactions that require different input and output COMMAREAs. The parameters (In, Out) are named as though they are being looked at from the host perspective; that is, the first (In) parameter is destined for the host; the second parameter (Out) comes from the host.) These should not refer to the same instance of the `IVJCicsEciCommArea`.

The size of the input COMMAREA must be equal to or larger than the size of the output COMMAREA, otherwise the run-time will throw an exception with message IVJC0085E. You can extend your input COMMAREA definition without affecting the processing, by adding a COBOL FILLER field to the end of the input COMMAREA, large enough to make the lengths match.

DFHCNV Support

It is possible to invoke transactions that use the DFHCNV macro to convert data between workstation and host formats.

The `IVJCicsUOWInterface` class now provides two properties that control conversions:

- ❑ `targetCodePage`—controls the format of the character data. For the transactions that use DFHCNV, the value of this property can be set to the code page that the client is expected to produce. Numeric-edited and numeric-display types (for example, PIC \$99, PIC 99999 DISPLAY) are treated as character strings for the purpose of this translation.

- ❑ **targetIntEndian**—controls the endian of 2- and 4-byte integers sent to the host. To cause no endian conversion to the MVS format, set this value to:

```
IVJCicsUOWInterface.UOW_LITTLE_ENDIAN
```

All other data types are translated on the workstation to the correct host format.

Closing the CICS Gateway for Java

The CICS Gateway for Java can now be closed directly by the program, through the `closeGateway` method. Invoking the following methods of the `IVJCicsUOWInterface` bean restores the connection:

```
invokeTxn(...)
asynchInvokeTxn(...)
```

Warning: `closeGateway` terminates the connection immediately. If any program threads are waiting for the results from the Gateway, they will return an error.

Setting the CICS Transaction ID

The CICS transaction ID (under which the application program runs) can now be set with the `transactionToInvoke` property. Methods to set and get the `transactionToInvoke` are:

```
void setTransactionToInvoke(String arg)
String getTransactionToInvoke()
```

Therefore, the `transactionToInvoke` property can be set in the `IVJCicsEciCommArea` bean (as an argument to either `invokeTxn` or `asynchInvokeTxn`) or in the `IVJCicsUOWInterface` (with `setTransactionToInvoke`). If the `transactionToInvoke` property is set in the `IVJCicsEciCommArea` bean by `invokeTxn` or `asynchInvokeTxn`, its value will be used, regardless of the value in the `IVJCicsUOWInterface`.

When the two argument versions of either the `invokeTxn` or `asynchInvokeTxn` method are used, only the first `IVJCicsEciCommArea` bean is looked at for this information. The order in which the transaction ID is used is as follows:

- ❑ `transactionToInvoke` of the `IVJCicsEciCommArea` object
- ❑ `transactionToInvoke` of the `IVJCicsUOWInterface` object
- ❑ default CICS transaction (CPMI), when none of the above is set

Specifying a Program to Execute

The specification of the program to be invoked can now be attached to either the `IVJCicsEciCommArea` or the `IVJCicsUOWInterface`, through the `progToCall` property. The access methods are:

```
void setProgToCall(String arg)
String getProgToCall()
```

If the `progToCall` property is set on the `IVJCicsEciCommArea` bean, passed as an argument to either `invokeTxn` or `asynchInvokeTxn`, its value will be used, regardless of the value in the `IVJCicsUOWInterface`. When the two argument versions of either the `invokeTxn` or `asynchInvokeTxn` method are used, only the first `IVJCicsEciCommArea` bean is looked at for this information. The order in which the `progToCall` is used is as follows:

- ☐ `progToCall` of the `IVJCicsEciCommArea` object
- ☐ `progToCall` of the `IVJCicsUOWInterface` object
- ☐ empty string, when none of the above is set

There are two ways to use the `IVJCicsUOWInterface`:

- ☐ As though it represents a single program acting (serially) on one or more `COMMAREA` parts
- ☐ As though it represents a CICS system, acting (serially) on one or more *program parts*. (A program part in this context refers to the `COMMAREA` data, plus optionally the program name to be used for the transaction ID.)

Either form can be used. The form you choose will depend on the nature of your application. If you are invoking multiple transactions on the host CICS system and want to keep the number of gateway connections to your Web server to a minimum, consider using a single `IVJCicsUOWInterface`. If the number of connections is not critical, you can use a separate `IVJCicsUOWInterface` for each invocation. If you are using the unit of work model (that is, you can commit or backout a series of program invocations), each member of the unit of work must be passed through the same `IVJCicsUOWInterface`.

To accommodate both styles, the CICS transaction ID and program name can be specified on either the `IVJCicsUOWInterface` or in the bean representing the `COMMAREA`. For each of these values, if the `COMMAREA` has a value specified, it will be used; if no value is specified, the `IVJCicsUOWInterface`'s value will be used.

The CICS Enterprise Access Builder COBOL parser requires a program name to be specified (either in the IDE GUI or as a mandatory argument at the command line). To use the value in the IVJCicsUOWInterface bean, the developer must clear the value of the program name (and transaction ID, if set) of the IVJCicsEciCommArea bean, so that it does not override the IVJCicsUOWInterface value(s). This can be done by calling the IVJCicsEciCommArea's setProgToCall (and setTransactionToInvoke, if appropriate), passing a blank or empty string (for example, setProgToCall("")).

Regardless of the form you elect to use, every IVJCicsUOWInterface will result in a socket connection to the server (or the Web server when the Java program is run as an applet) where the server side of the CICS Java Gateway is running. This connection will remain open indefinitely. Again, depending on the nature of your application, you may want to close the connection after some situation has occurred (timer interrupt, button clicked), rather than waiting for the application to end (thus destroying the IVJCicsUOWInterface and gateway). To close the connection, call IVJCicsUOWInterface.closeGateway. This does not end the IVJCicsUOWInterface, and subsequent reuse of invokeTxn or asyncInvokeTxn causes the gateway to be reopened.

Changes to IVJCicsEciCommArea Bean

The communications area bean has been improved in several ways.

Setting the CICS Transaction ID

The CICS transaction ID (under which the application program runs) can now be set with the transactionToInvoke property. Methods to set and get the transactionToInvoke:

```
void setTransactionToInvoke(String arg)
String getTransactionToInvoke()
```

If this property is set, its value will be used, regardless of the value in the IVJCicsUOWInterface. The order in which the transaction ID is used is as follows:

- ❑ transactionToInvoke of the IVJCicsEciCommArea object
- ❑ transactionToInvoke of the IVJCicsUOWInterface object
- ❑ default CICS transaction (CPMI), when none of the above is set

Double-Byte Character Set Support

This release of CICS Enterprise Access Builder includes support for DBCS characters. The imported COBOL transactions can contain the following COBOL data types:

PIC G(nn) USAGE DISPLAY-1. PIC N(nn).

Because these types can only contain double byte characters, an exception with message IVJC0086E will be thrown if an attempt is made to pass single-byte characters in the string corresponding to one of the above COBOL types. The VALUE clause can be used with PIC G(nn) and PIC N(nn) data types subject to the limitations described in the product documentation's Limitations section.

JDK restriction: Due to JDK limitations, the supported DBCS does not include 37 'itaiji' characters. See the JDK documentation for further information.

Changes to Limitation on Code Page

The target host code page is no longer restricted to IBM-037. It can be any of the code pages supported by the JDK.

C++ Access Builder

There are a number of changes in the C++ Access Builder.

Compatibility between Versions

The 1.0.1 NL version of the C++ Access Builder run-time library (ivjdjs10.dll) and the C++ Access Builder Java class library (ivjeab.zip) support code generated using the 1.0 or 1.0.1 NL C++ Access Builder importer (ivj2cpp.exe).

The 1.0 version of the C++ Access Builder runtime library only supports code generated using the 1.0 version of the C++ Access Builder importer.

When installing a DLL built with the 1.0.1 NL version of the C++ Access Builder, ensure that it picks up the correct class library (with the CLASSPATH environment variable) and the correct runtime library (with PATH on Windows, and BEGINPATH or LIBPATH on OS/2).

Deleting C++ Objects Allocated in Java

A native delete method has been added to the parent class of all generated Java representative classes. In order for Java to free the space used by a C++ object that was allocated by a new method on the Java representative object, the delete method must be called.

For example, if you had a C++ class named *Person*, which has been wrapped, its Java representative class would be *Person*. In Java, if you had this code:

```
Person x = new Person("John Doe");
```

it would allocate a C++ *Person* object to which *x* would point. Because this C++ object was allocated in Java, to free the C++ object you must at some point in your Java code have this code:

```
x.delete();
```

When the delete method has been invoked, the C++ object will be deleted, and you should no longer reference the variable *x*.

Character Arrays

Only character arrays of type `wchar_t` will support DBCS characters. Although a Java `char` array can hold DBCS characters, when mapped to C++ each Java character is mapped to a C++ character.

Because a C++ character is 1 byte, the flow from Java to C++ is constrained to single-byte character set for the types `char`, `unsigned char`, and `signed char`.

Signed Characters

Signed characters (`SCHAR`) are treated as a byte. Therefore, there is no code page conversion when an `SCHAR` is constructed or set using a Java byte.

Pointers

In Version 1.0, the generated set method for pointers to primitive types was setting by value instead of by address. The corresponding get method was getting by address.

Since the `P<PrimitiveType>` classes already contain methods for getting and setting by value, the generated set method in Version 1.0.1 sets by address.

With Version 1.0.1 the generated set method in the C++ wrapper files for the types `PSCHAR`, `PUCHAR`, and `PWCHAR` compile properly.

Exceptions

As part of the support for national languages, conversion to different code pages has been added. Code page conversion has caused more methods in the C++ Access Builder library to throw an exception. For many classes in the C++ Access Builder library, the `toString` method throws a `Java.lang.IllegalArgumentException`. In all other cases where an exception was added, the exception thrown is `IVJJException`. Therefore you have to add try and catch blocks or rethrow the `IVJJException`.

Migration to Version 1.0.1?

You may decide not to migrate immediately to Version 1.0.1 because:

- ☐ You do not need other language support
- ☐ You consider that Version 1.0.1 has a larger memory requirement and decide that Version 1.0 is good enough

We determined that the additional features of Version 1.0.1 did not offer us significant benefits and decided to use Version 1.0 for our sample applications.



Special Notices

This publication is intended to help application developers create Java-based enterprise applications. The information in this publication is not intended as the specification of any programming interfaces that are provided by VisualAge for Java Enterprise. See the PUBLICATIONS section of the IBM Programming Announcement for VisualAge for Java for more information about what publications are considered to be product documentation.

References in this publication to IBM products, programs or services do not imply that IBM intends to make these available in all countries in which IBM operates.

Any reference to an IBM product, program, or service is not intended to state or imply that only IBM's product, program, or service may be used. Any functionally equivalent program that does not infringe any of IBM's intellectual property rights may be used instead of the IBM product, program or service.

This document has not been subjected to any formal review and has not been checked for technical accuracy. Results may be individually evaluated for applicability to a particular installation. You may discuss pertinent information from this document with a

customer, and you may abstract pertinent information for presentation to your customers. However, any code included is for internal information purposes only and may not be given to customers. If included code is identified as incidental programming, its use must conform to the guidelines in the relevant section of the sales manual.

References in this publication to IBM products, programs or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program, or service is not intended to state or imply that only IBM's product, program, or service may be used. Any functionally equivalent program that does not infringe any of IBM's intellectual property rights may be used instead of the IBM product, program or service.

Information in this book was developed in conjunction with use of the equipment specified, and is limited in application to those specific hardware and software products and levels.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to the IBM Director of Licensing, IBM Corporation, 500 Columbus Avenue, Thornwood, NY 10594 USA.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact IBM Corporation, Dept. 600A, Mail Drop 1329, Somers, NY 10589 USA.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The information contained in this document has not been submitted to any formal IBM test and is distributed AS IS. The information about non-IBM ("vendor") products in this manual has been supplied by the vendor and IBM assumes no responsibility for its accuracy or completeness. The use of this information or the implementation of any of these techniques is a customer responsibility and depends on the customer's ability to evaluate and integrate them into the customer's operational environment. While each item may have been reviewed by IBM for accuracy in a specific situation, there is no guarantee that the same or similar results will be obtained elsewhere. Customers attempting to adapt these techniques to their own environments do so at their own risk.

Any performance data contained in this document was obtained in a controlled environment based on the use of specific data and is presented only to illustrate techniques and procedures to assist IBM personnel to better understand IBM products. The results that may be obtained in other operating environments may vary significantly. Users of this document should verify the applicable data in their specific environment. No performance data may be abstracted or reproduced and given to non-IBM personnel without prior written approval by Business Practices.

Any performance data contained in this document was determined in a controlled environment, and therefore, the results that may be obtained in other operating environments may vary significantly. Users of this document should verify the applicable data for their specific environment.

The following document contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples contain the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

Reference to PTF numbers that have not been released through the normal distribution process does not imply general availability. The purpose of including these reference numbers is to alert IBM customers to specific information relative to the implementation of the PTF when it becomes available to each customer according to the normal IBM PTF distribution process.

You can reproduce a page in this document as a transparency, if that page has the copyright notice on it. The copyright notice must appear on each page being reproduced.

The following terms are trademarks of the International Business Machines Corporation in the United States and/or other countries.

IBM	CICS
DB2	MVS
OS/2	OS/390
S/390	ThinkPad
VisualAge	

The following terms are trademarks of other companies:

C-bus is a trademark of Corollary, Inc.

Java and HotJava are trademarks of Sun Microsystems, Incorporated.

Microsoft, Windows, Windows NT, and the Windows 95 logo are trademarks or registered trademarks of Microsoft Corporation.

PC Direct is a trademark of Ziff Communications Company and is used by IBM Corporation under license.

Pentium, MMX, ProShare, LANDesk, and ActionMedia are trademarks or registered trademarks of Intel Corporation in the U.S. and other countries.

UNIX is a registered trademark in the United States and other countries licensed exclusively through X/Open Company Limited.

Other company, product, and service names may be trademarks or service marks of others.

A large, white, sans-serif capital letter 'D' is positioned on the left side of the page. It is set against a solid gray square background that is partially visible behind the main title text.

Related Publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this redbook.

International Technical Support Organization Publications

For information on ordering these ITSO publications see “How To Get ITSO Redbooks” on page 391.

- ❑ *Programming with VisualAge for Java*, SG24-2232, Prentice Hall, 1998, ISBN 0-13-911371-1
- ❑ *VisualAge for Java Enterprise Team Support*, SG24-5245 (July 1998)
- ❑ *Creating Java Applications with NetRexx*, SG24-2216
- ❑ *Unlimited Enterprise Access with Java and VisualAge Generator*, SG24-5246 (June 1998)
- ❑ *VisualAge Generator Client/Server Communications*, SG24-4237
- ❑ *VisualAge Generator Version 3.0 System Development Guide*, SG24-4230
- ❑ *CBConnector Overview*, SG24-2022
- ❑ *CBConnector Cookbook Volume 1*, SG24-2033
- ❑ *Connecting the Enterprise to the Internet with MQSeries and VisualAge for Java*, SG24-2144
- ❑ *Factoring JavaBeans in the Enterprise*, SG24-5051
- ❑ *JavaBeans by Example, Cooking with Beans in the Enterprise*, SG24-2035, Prentice Hall, 1997
- ❑ *Java Network Security*, SG24-2109, Prentice Hall, 1998
- ❑ *Building AS/400 Applications with Java*, SG24-2163
- ❑ *Accessing the AS/400 System with Java*, SG24-2152
- ❑ *World Wide Web Programming: VisualAge for C++ and Smalltalk*, SG24-4734, Prentice Hall, 1997
- ❑ *Programming with VisualAge for C++ for Windows*, SG24-4782, Prentice Hall, 1997
- ❑ *Object-Oriented Application Development with VisualAge for C++ for OS/2*, SG24-2593, Prentice Hall, 1996
- ❑ *VisualAge and Transaction Processing in a Client/Server Environment*, GG24-4487, Prentice Hall, 1995

Redbooks on CD-ROMs

Redbooks are also available on CD-ROMs. **Order a subscription** and receive updates 2-4 times a year at significant savings.

CD-ROM Title	Subscription Number	Collection Kit Number
System/390 Redbooks Collection	SBOF-7201	SK2T-2177
Networking and Systems Management Redbooks Collection	SBOF-7370	SK2T-6022
Transaction Processing and Data Management Redbook	SBOF-7240	SK2T-8038
Lotus Redbooks Collection	SBOF-6899	SK2T-8039
Tivoli Redbooks Collection	SBOF-6898	SK2T-8044
AS/400 Redbooks Collection	SBOF-7270	SK2T-2849
RS/6000 Redbooks Collection (HTML, BkMgr)	SBOF-7230	SK2T-8040
RS/6000 Redbooks Collection (PostScript)	SBOF-7205	SK2T-8041
RS/6000 Redbooks Collection (PDF Format)	SBOF-8700	SK2T-8043
Application Development Redbooks Collection	SBOF-7290	SK2T-8037

Other Publications

These publications are also relevant as further information sources:

- ❑ *Developing JavaBeans Using VisualAge for Java*, by Dale Nilsson and Peter Jakab. John Wiley & Sons, 1998. ISBN 0-471-29788-7.
- ❑ *Database Programming with JDBC and Java 1.1*, SR23-8029, by George Reese. O'Reilly, 1997.
- ❑ *Java Database Programming with JDBC*, SR23-8101, by Pratik Patel and Karl Moss. Coriolis, 1997.
- ❑ *JDBC Database Access with Java, A Tutorial and Annotated*, SR23-8106, by Graham Hamilton, Rick Cattell, and Maydene Fisher. Addison-Wesley, 1997.
- ❑ *Client/Server Programming with Java and CORBA*, SR23-7787, by Robert Orfali and Dan Harkey. John Wiley & Sons, 1997.
- ❑ *Java Network Programming*, SR23-7798, by Eliotte Rusty Harold. O'Reilly, 1997.
- ❑ *Advanced Java Networking*, SR23-8034, by Prashant Sridharan. Prentice Hall, 1997.
- ❑ *Java 1.1 Networking and Communications*, SR23-8126, by Todd Curtis. Prentice Hall, 1997.
- ❑ *Enterprise Java*, SR23-8198, by Jaff Savit, Sean Wilcox, and Bhuvana Jayaraman. McGraw-Hill, 1997.
- ❑ *Java Distributed Computing*, SR23-8316, by Jim Farley. O'Reilly, 1998.
- ❑ *Core Java 1.1, Volume II - Advanced Features*, SR23-8311, by Cay S. Horstmann and Gary Cornell. Prentice Hall, 1998.
- ❑ *Developing Java Beans*, SR23-8021, by Robert Englander. O'Reilly, 1997.

How To Get ITSO Redbooks

This section explains how both customers and IBM employees can find out about ITSO redbooks, CD-ROMs, workshops, and residencies.

This information was current at the time of publication, but is continually subject to change. The latest information may be found at URL <http://www.redbooks.ibm.com>.

How IBM Employees Can Get ITSO Redbooks

Employees may request ITSO deliverables (redbooks, BookManager BOOKs, and CD-ROMs) and information about redbooks, workshops, and residencies in the following ways:

- ❑ **PUBORDER** — to order hard copies in United States

- ❑ **GOPHER link to the Internet**
- type GOPHER.WTSCPOK.ITSO.IBM.COM

- ❑ **Tools disks**

To get LIST3820s of redbooks, type:

```
TOOLS SENDTO EHONE4 TOOLS2 REDPRINT GET SG24xxxx PACKAGE  
TOOLS SENDTO CANVM2 TOOLS REDPRINT GET SG24xxxx PACKAGE (Canadian users only)
```

To get lists of redbooks:

```
TOOLS SENDTO USDIST MKTTOOLS MKTTOOLS GET ITSOCAT TXT
```

To register for information on workshops, residencies, and redbooks:

```
TOOLS SENDTO WTSCPOK TOOLS ZDISK GET ITSOREGI 1996
```

For a list of product area specialists in the ITSO:

```
TOOLS SENDTO WTSCPOK TOOLS ZDISK GET ORGCARD PACKAGE
```

- ❑ **Redbooks Web Site on the World Wide Web**

<http://w3.itso.ibm.com/redbooks>

- ❑ **IBM Direct Publications Catalog on the World Wide Web**

<http://www.elink.ibm.link.ibm.com/pbl/pbl>

IBM employees may obtain LIST3820s of redbooks from this page.

- ❑ **REDBOOKS category on INEWS**

- ❑ **Online** — send orders to: USIB6FPL at IBMMAIL or DKIBMBSH at IBMMAIL

- ❑ **Internet Listserver** — With an Internet E-mail address, anyone can subscribe to an IBM Announcement Listserver. To initiate the service, send an E-mail note to announce@webster.ibm.link.ibm.com with the keyword subscribe in the body of the note (leave the subject line blank). A category form and detailed instructions will be sent to you.

How Customers Can Get ITSO Redbooks

Customers may request ITSO deliverables (redbooks, BookManager BOOKs, and CD-ROMs) and information about redbooks, workshops, and residencies in the following ways:

- ❑ **Online Orders** (Do not send credit card information over the Internet)—send orders to:

	IBMMAIL	Internet
In United States:	usib6fpl at ibmmail	usib6fpl@ibmmail.com
In Canada:	caibmbkz at ibmmail	lmannix@vnet.ibm.com
Outside North America:	dkibmbsh at ibmmail	bookshop@dk.ibm.com

- ❑ **Telephone orders**

United States (toll free)	1-800-879-2755	United States (toll free)
Canada (toll free)	1-800-IBM-4YOU	Canada (toll free)
Outside North America	(long distance charges apply)	Outside North America
(+45) 4810-1320 - Danish	(+45) 4810-1020 - German	(+45) 4810-1320 - Danish
(+45) 4810-1420 - Dutch	(+45) 4810-1620 - Italian	(+45) 4810-1420 - Dutch
(+45) 4810-1540 - English	(+45) 4810-1270 - Norwegian	(+45) 4810-1540 - English
(+45) 4810-1670 - Finnish	(+45) 4810-1120 - Spanish	(+45) 4810-1670 - Finnish
(+45) 4810-1220 - French	(+45) 4810-1170 - Swedish	(+45) 4810-1220 - French

- ❑ **Mail Orders** — send orders to:

IBM Publications	IBM Publications	IBM Direct Services
Publications Customer Support	144-4th Avenue, S.W.	Sortemosevej 21
P.O. Box 29570	Calgary, Alberta T2P 3N5	DK-3450 Allerød
Raleigh, NC 27626-0570	Canada	Denmark
USA		

- ❑ **Fax** — send orders to:

United States (toll free)	1-800-445-9269	United States (toll free)
Canada	1-403-267-4455	Canada
Outside North America	(+45) 48 14 2207	Outside North America
	(long distance charge)	

- ❑ **1-800-IBM-4FAX (United States) or (+1) 415 855 43 29 (Outside USA)** — ask for:

Index # 4421 Abstracts of new redbooks
Index # 4422 IBM redbooks
Index # 4420 Redbooks for last six months

- ❑ **Direct Services** - send note to softwareshop@vnet.ibm.com

- ❑ **On the World Wide Web**

Redbooks Home Page <http://www.redbooks.ibm.com>
IBM Direct Publications Catalog <http://www.elink.ibm.link.ibm.com/pbl/pbl>

- ❑ **Internet Listserver**

With an Internet E-mail address, anyone can subscribe to an IBM Announcement List-server. To initiate the service, send an E-mail note to announce@webster.ibm.link.ibm.com with the keyword `subscribe` in the body of the note (leave the subject line blank).

IBM Redbook Order Form

Please send me the following:

Title	Order Number	Quantity
-------	--------------	----------

First name

Last name

Company

Address

City

Postal code

Country

Telephone number

Telefax number

VAT number

☐ Invoice to customer number

☐ Credit card number

Credit card expiration date

Card issued to

Signature

We accept American Express, Diners, Eurocard, Master Card, and Visa. Payment by credit card not available in all countries. Signature mandatory for credit card payment.

Glossary

This glossary defines terms and abbreviations that are used in this book. If you do not find the term you are looking for, refer to the *IBM Dictionary of Computing*, New York: McGraw-Hill, 1994.

This glossary includes terms and definitions from the *American National Standard Dictionary for Information Systems*, ANSI X3.172-1990, copyright 1990 by the American National Standards Institute (ANSI). Copies may be purchased from the American National Standards Institute, 1430 Broadway, New York, New York 10018.

A

abstract class. A class that provides common behavior across a set of subclasses but is not itself designed to have instances that work. An abstract class represents a concept; classes derived from it represent implementations of the concept. See also *base class*.

access application. Generated by the Data Access Builder for each schema mapping, an executable GUI that provides access to the database using the other classes generated for the mapping.

accessor methods. Methods that an object provides to define the interface to its instance variables. The accessor method to return the value of an instance variable is called a *get* method or *getter* method, and the accessor method to assign a value to an instance variable is called a *set* method or *setter* method.

applet. A Java program designed to run within a Web browser. Contrast with application.

application. In Java programming, a self-contained, stand-alone Java program that includes a *main()* method. Contrast with applet.

argument. A data element, or value, included as a bean in a method call. Arguments provide additional information that the called method can use to perform the requested operation.

attribute. A specification of a property of a bean. For example, a customer bean could have a name attribute and an address attribute. An attribute can itself be a bean with its own behavior and attributes. In the Data Access Builder, the aspect of a schema mapping that represents a column in a database table.

B

base class. A class from which other classes or beans are derived. A base class may itself be derived from another base class. See also *abstract class*.

bean. A definition or instance of a JavaBeans component. See also *JavaBeans*.

BeanInfo. (1) A companion class for a bean that defines a set of methods that can be accessed to retrieve information on the bean's properties, events, and methods. (2) In the VisualAge for Java IDE, a page in the class browser that provides bean information.

beans palette. In the Visual Composition Editor, a two-column pane that contains prefabricated beans that you can select and manipulate to create programs. The left column contains categories of beans, and the right column contains beans for the selected category. The default set of beans generally represents JDK AWT components. You can add your own categories and beans to the beans palette.

break point. A point in a computer program where the execution can be halted.

browser. (1) In VisualAge for Java, a window that provides information on program elements. There are browsers for projects, packages, classes, methods, and interfaces. (2) An Internet-based tool that lets user browse Web sites.

C

C++ Access Builder. A VisualAge for Java Enterprise tool that generates beans to access C and C++ DLLs.

category. In the Visual Composition Editor, a selectable grouping of beans represented by an icon in the left-most column. Selecting a category displays the beans belonging to that category in the next column. See also *beans palette*.

CICS Access Builder. A VisualAge for Java Enterprise tool that generates beans to access CICS transactions through the CICS Gateway for Java and CICS Client.

CICS Client. A server program that processes CICS ECI calls, forwarding transaction requests to a CICS program running on a host.

CICS ECI. An API that provides C and C++ programs with procedural access to transactions.

CICS Gateway for Java. A server program that processes Java ECI calls and forwards CICS ECI calls to the CICS Client.

class. An aggregate that defines properties, operations, and behavior for all instances of that aggregate.

class hierarchy. The relationships between classes that share a single inheritance. All Java classes inherit from the Object class.

class library. A collection of classes.

class method. See *method*.

CLASSPATH. In your deployment environment, the environment variable that specifies the directories in which to look for class and resource files.

client/server. The model of interaction in distributed data processing where a program at one location sends a request to a program at another location and awaits a response. The requesting program is called a *client*, and the answering program is called a *server*.

client-side server proxy. Generated by the RMI Access Builder, a local representative of a remote bean. This proxy provides access to the operations of the server bean, allowing a Java client to work with it as if it were the server bean. See also *proxy bean* and *server-side server proxy*.

Class Browser. In the VisualAge for Java IDE, a tool used to browse the classes loaded in the workspace.

collection. A set of features in which each feature is an object.

commit. The operation that ends a unit of work and updates the database such that other processes can access any changes made.

Common Object Request Broker Architecture (CORBA). A middleware specification which defines a software bus—the Object Request Broker (ORB)—that provides the infrastructure.

communications area (COMMAREA). In a CICS transaction program, a group of records that describes both the format and volume of data used.

component model. An architecture and an API that allows developers to define reusable segments of code that can be combined to create a program. VisualAge for Java uses the JavaBeans component model.

composite bean. A bean that is composed of a bean and one or more subbeans. A composite bean can contain visual beans, nonvisual beans, or both. See also *nonvisual bean*, *bean*, and *visual bean*.

concrete class. A subclass of an abstract class that is a specialization of the abstract class.

connection. In the Visual Composition Editor, a visual link between two components that represents the relationship between the components. Each connection has a source, a target, and other properties. See also *event-to-method connection*, *event-to-property con-*

nection, parameter connection, property-to-method connection, and property-to-property connection.

console. In VisualAge for Java, the window that acts as the standard input (System.in) and standard output (System.out) device for programs running in the VisualAge for Java IDE.

construction from parts. A software development technology in which applications are assembled from existing and reusable software components, known as *parts*. In VisualAge for Java, parts are called *beans*.

constructor. A special class method that has the same name as the class and is used to construct and possibly initialize objects of its class type.

container. A component that can hold other components. In Java, examples of containers include applets, frames, and dialogs. In the Visual Composition Editor, containers can be graphically represented and generated.

current edition. The edition of a program element that is currently in the workspace. See also *open edition*.

cursor. A database control structure used by the Data Access Builder to point to a specific row within some ordered set of rows and to retrieve rows from a set, possibly making updates or deletions.

D

data abstraction. A data type with a private representation and a public set of operations. The Java language uses the concept of classes to implement data abstraction.

Data Access Builder. A VisualAge for Java Enterprise tool that generates beans to access and manipulate the content of JDBC/ODBC-compliant relational databases.

DB2 for MVS/ESA. An IBM relational database management system for the MVS operating system.

double-byte character set (DBCS). A set of characters in which each character is represented by 2 bytes. Languages such as Japanese, Chinese, and Korean, which contain more symbols than can be represented by 256 code points, require double-byte character sets. Compare with *single-byte character set*.

dynamic link library (DLL). A file containing executable code and data bound to a program at run time rather than at link time. The C++ Access Builder generates beans and C++ wrappers that let your Java programs access C++ DLLs.

E

edition. A specific “cut” of a program element. VisualAge for Java supports multiple editions of program elements. See also *current edition, open edition, and versioned edition*.

encapsulation. The hiding of a software object's internal representation. The object provides an interface that queries and manipulates the data without exposing its underlying structure.

enterprise access builders. In VisualAge for Java Enterprise, a set of code-generation tools. See also *C++ Access Builder, CICS Access Builder, Data Access Builder, and RMI Access Builder*.

event. An action by a user program, or a specification of a notification that may trigger specific behavior. In JDK 1.1, events notify the relevant listener classes to take appropriate actions.

event-to-method connection. A connection from an event generated by a bean to a method of another bean. When the connected event occurs, the method is executed. See also *connection*.

event-to-property connection. A connection that changes the value of a property when a certain event occurs. See also *connection*.

F

feature. (1) A major component of a software product that can be installed separately. (2) In VisualAge for Java, a method, field, or event that is available from a bean's interface and to which other beans can connect.

field. A data object in a class. For example, a customer class could have a name field and an address field. A field can itself be an object with its own behavior and fields. By default, a field, in contrast to a property, does not support event notification.

free-form surface. The large open area of the Visual Composition Editor where you can work with visual and nonvisual beans. You add, remove, and connect beans on the free-form surface.

framework. A set of cooperative classes with strong connections that provide a template for development.

G

garbage collection. A Smalltalk process for periodically identifying unreferenced objects and deallocating their memory.

gateway. A host computer that connects networks that communicate in different languages. For example, a gateway connects a company's LAN to the Internet.

graphical user interface (GUI). A type of interface that enables users to communicate with a program by manipulating graphical features, rather than by entering commands. Typically, a graphical user interface includes a combination of graphics, pointing devices, menu bars and other menus, overlapping windows, and icons.

H

hypertext. Text in a document that contains a hidden link to other text. You can click a mouse on a hypertext word and it will take you to the text designated in the link.

Hypertext is used in Windows help programs and CD encyclopedias to jump to related references elsewhere within the same document. The wonderful thing about hypertext, however, is its ability to link—using HTTP over the Web—to any Web document in the world, with only a single mouse click.

Hypertext Markup Language (HTML).

The basic language that is used to build hypertext documents on the World Wide Web. It is used in basic, plain ASCII-text documents, but when those documents are interpreted (called rendering) by a Web browser such as Netscape, the document can display formatted text, color, a variety of fonts, graphic images, special effects, hypertext jumps to other Internet locations, and information forms.

Hypertext Transfer Protocol (HTTP).

The protocol for moving hypertext files across the Internet. Requires an HTTP client program on one end, and an HTTP server program on the other end. HTTP is the most important protocol used in the World Wide Web.

I

inheritance. (1) A mechanism by which an object class can use the attributes, relationships, and methods defined in more abstract classes related to it (its base classes). (2) An object-oriented programming technique that allows you to use existing classes as bases for creating other classes.

instance. Synonym for *object*, a particular instantiation of a data type.

Integrated Development Environment (IDE).

In VisualAge for Java, the set of windows that provide the user with access to development tools. The primary windows are Workbench, Log, Console, Debugger, and Repository Explorer.

interchange file. A file that you can export from VisualAge for Java that contains information about selected projects or packages. This file can then be imported into any VisualAge for Java session.

interface. A set of methods that can be accessed by any class in the class hierarchy. The Interface page in the Workbench lists all interfaces in the workspace.

Internet. The vast collection of interconnected networks that use TCP/IP protocols and evolved from the ARPANET of the late 1960s and early 1970s.

intranet. A private *network*, inside a company or organization, that uses the same kinds of software that you would find on the public *Internet*. Many of the tools used on the Internet are being used in private networks; for example, many companies have Web servers that are available only to employees.

Internet Protocol (IP). The rules that provide basic Internet functions. See *Transmission Control Protocol/Internet Protocol*.

IP number. An Internet address that is a unique number consisting of four parts separated by dots, sometimes called a *dotted quad* (for example: 198.204.112.1). Every Internet computer has an IP number, and most computers also have one or more domain names that are plain language substitutes for the dotted quad.

J

Java. A new programming language invented by Sun Microsystems that is specifically designed for writing programs that can be safely downloaded to your computer through the Internet and immediately run without fear of viruses or other harm to your computer or files. Using small Java programs (called *applets*), Web pages can include functions such as animations, calculators, and other fancy tricks. We can expect to see a huge variety of features added to the Web through Java, because you can write a Java program to do almost anything a regular computer program can do and then include that Java program in a Web page.

Java archive (JAR). A platform-independent file format that groups many files into one. JAR files are used for compression,

reduced download time, and security. Because the JAR format is written in Java, JAR files are fully extensible.

JavaBeans. In JDK 1.1, the specification that defines the platform-neutral component model used to represent parts. Instances of JavaBeans (often called beans) may have methods, properties, and events.

Java Database Connectivity (JDBC). In JDK 1.1, the specification that defines an API that enables programs to access databases that comply with this standard.

Java Native Interface (JNI). In JDK 1.1, the specification that defines a standard naming and calling convention so that the Java virtual machine can locate and invoke methods written in a language different from Java. See also ***native method***.

K

keyword. A predefined word reserved for Java, that may not be used as an identifier.

L

legacy code. Existing code that a user might have. Legacy applications often have character-based, nongraphical user interfaces. Usually they are written in a non-object-oriented language, such as C or COBOL.

listener. In JDK 1.1, a class that receives and handles events.

local area network (LAN). A computer network located on a user's establishment within a limited geographical area. A LAN typically consists of one or more server machines providing services to a number of client workstations.

log. In VisualAge for Java, the window that displays messages and warnings during development.

M

mapping. See *schema mapping*.

member. (1) A data object in a structure or a union. (2) In Java, classes and structures can also contain functions and types as members.

method. A fragment of Java code within a class that can be invoked and passed a set of parameters to perform a specific task.

method call. A communication from one object to another that requests the receiving object to execute a method. A method call consists of a method name that indicates the requested method and the arguments to be used in executing the method. The method call always returns some object to the requesting object as the result of performing the method. Synonym for *message*.

message. A request from one object that the receiving object implement a method. Because data is encapsulated and not directly accessible, a message is the only way to send data from one object to another. Each message specifies the name of the receiving object, the method to be implemented, and any arguments the method needs for implementation. Synonym for *method call*.

model. A nonvisual bean that represents the state and behavior of an object, such as a customer or an account. Contrast with *view*.

N

native method. Method written in a language other than Java that can be called by a Java object through the JNI specification.

named package. In the VisualAge for Java IDE, a package that has been explicitly named and created.

nonvisual bean. In the Visual Composition Editor, a bean that has no visual representation at run time. A nonvisual bean typically represents some real-world object that exists in the business environment. Compare with *model*. Contrast with *view* and *visual bean*.

notification framework. In JDK 1.1, a set of classes that implement the *notifier/listener* protocol. The notification framework is the base of the construction from beans technology (Visual Composition Editor).

O

object. (1) A computer representation of something that a user can work with to perform a task. An object can appear as text or an icon. (2) A collection of data and methods that operate on that data, which together represent a logical entity in the system. In object-oriented programming, objects are grouped into classes that share common data definitions and methods. Each object in the class is said to be an instance of the class. (3) An instance of an object class consisting of attributes, a data structure, and operational methods. It can represent a person, place, thing, event, or concept. Each instance has the same properties, attributes, and methods as other instances of the object class, although it has unique values assigned to its attributes.

object class. A template for defining the attributes and methods of an object. An object class can contain other object classes. An individual representation of an object class is called an *object*.

object factory. A nonvisual bean capable of dynamically creating new instances of a specified bean. For example, during the execution of an application, an object factory can create instances of a new class to collect the data being generated.

object-oriented programming (OOP). A programming approach based on the concepts of data abstraction and inheritance. Unlike procedural programming techniques, object-oriented programming concentrates on those data objects that constitute the problem and how they are manipulated, not on how something is accomplished.

Object Request Broker (ORB). A CORBA term designating the means by which objects transparently make requests and receive responses from objects, whether they are local or remote.

ODBC driver. An ODBC driver is a DLL that implements ODBC function calls and interacts with a data source.

Open Database Connectivity (ODBC). A Microsoft developed C database API that allows access to database management systems calling callable SQL, which does not require the use of an SQL preprocessor. In addition, ODBC provides an architecture that allows users to add modules (database drivers) that link the application to their choice of database management systems at run time. Applications no longer need to be directly linked to the modules of all the database management systems that are supported.

open edition. An edition of a program element that can still be modified; that is, the edition has not been versioned. An open edition may reside in the workspace as well as in the repository.

operation. A method or service that can be requested of an object.

overloading. An object-oriented programming technique that allows redefinition of methods when the methods are used with class types.

P

package. A program element that contains related classes and interfaces.

palette. See *beans palette*.

parameter connection. A connection that satisfies a parameter of an action or method by supplying either a property's value or the return value of an action, method, or script. The parameter is always the source of the connection. See also *connection*.

parent class. The class from which another bean or class inherits data, methods, or both.

part. An existing, reusable software component. In VisualAge for Java, all parts created with the Visual Composition Editor conform to the JavaBeans component model and are

referred to as beans. See also *nonvisual bean* and *visual bean*. Compare with *Class Editor* and *Composition Editor*.

primitive bean. A basic building block of other beans. A primitive bean can be relatively complex in terms of the function it provides.

private. In Java, an access modifier associated with a class member. It allows only the class itself to access the member.

process. A collection of code, data, and other system resources, including at least one thread of execution, that performs a data processing task.

program. In VisualAge for Java, a term that refers to both Java applets and applications.

project. In VisualAge for Java, the topmost kind of program element. A project contains Java packages.

promote features. Make features of a subbean available to be used for making connections. This applies to subbeans that are to be included in other beans, for example, a subbean consisting of three push buttons on a panel. If this sample subbean is placed in a frame, the features of the push buttons would have to be promoted to make them available from within the frame.

property. An initial setting or characteristic of a bean; for example, a name, font, text, or positional characteristic.

property sheet. In the Visual Composition Editor, a set of name-value pairs that specify the initial appearance and other bean characteristics. A bean's property sheet can be viewed from the Properties secondary window.

property-to-method connection. A connection that calls a method whenever a property's value changes. It is similar to an event-to-method connection because the property's event ID is used to notify the method when the value of the property changes. See also *connection*.

property-to-property connection. A connection from a property of one bean to a property of another bean. When one property is updated, the other property is updated automatically. See also *connection*.

property-to-method connection. A connection from a property of a bean to a method. When the property undergoes a state change, the method is called. See also *connection*.

protected. In Java, an access modifier associated with a class member. It allows the class itself, subclasses, and all classes in the same package to access the member.

protocol. (1) The set of all messages to which an object will respond. (2) Specification of the structure and meaning (the semantics) of messages that are exchanged between a client and a server. (3) Computer rules that provide uniform specifications so that computer hardware and operating systems can communicate. It is similar to the way that mail, in countries around the world, is addressed in the same basic format so that postal workers know where to find the recipient's address, the sender's return address, and the postage stamp. Regardless of the underlying language, the basic protocols remain the same.

prototype. A method declaration or definition that includes both the return type of the method and the types of its arguments.

proxy-bean. A group of client-side and server-side objects that represent a remote server bean. The top-level class that implements the proxy bean is the client-side server proxy. See also *client-side server proxy* and *server-side server proxy*.

R

Remote Method Invocation (RMI). In JDK 1.1, the API that enables you to write distributed Java programs, allowing methods of remote Java objects to be accessed from other Java virtual machines.

remote object instance manager. Creates and manages instances of RMI server beans through their associated server-side server proxies.

repository. In VisualAge for Java, the storage area, separate from the workspace, that contains all editions (both open and versioned) of all program elements that have ever been in the workspace, including the current editions that are in the workspace. You can add editions of program elements to the workspace from the repository.

Repository Explorer. In VisualAge for Java, the window from which you can view and compare editions of program elements that are in the repository.

resource file. A noncode file that can be referred to from your Java program in VisualAge for Java. Examples include graphic and audio files.

RMI Access Builder. A VisualAge for Java Enterprise tool that generates proxy beans and associated classes and interfaces so you can distribute code for remote access, enabling Java-to-Java solutions.

RMI compiler. The compiler that generates stub and skeleton files that facilitate RMI communication. This compiler can be automatically invoked by the RMI Access Builder or from the Tools menu item.

RMI registry. A server program that allows remote clients to get a reference to a server bean.

roll back. The process of restoring data changed by SQL statements to the state at its last commit point.

S

schema. In the Data Access Builder, the representation of the database that will be mapped.

schema mapping. In the Data Access Builder, a set of definitions for all attributes matching all columns for your database table, view, or SQL statement. The mapping

contains the information required by the Data Access Builder to generate Java classes.

Scrapbook. In VisualAge for Java, the window from which you can write and test fragments of code, without having to define an encompassing class or method.

server. A computer that provides services to multiple users or workstations in a network; for example, a file server, a print server, or a mail server.

server bean. The bean that is distributed using RMI services and deployed on a server.

server-side server proxy. Generated by the RMI Access Builder, a companion class to the client-side server proxy, facilitating client-side server proxy communication over RMI. See also *client-side server proxy* and *proxy bean*.

service. A specific behavior that an object is responsible for exhibiting.

single-byte character set. A set of characters in which each character is represented by a 1-byte code.

SmartGuide. In IBM software products, an interface that guides you through performing common tasks.

SQL predicate. The conditional part of an SQL statement.

sticky. In the Visual Composition Editor, the mode that enables an application developer to add multiple beans of the same class (for example, three push buttons) without going back and forth between the beans palette and the free-form surface.

stored procedure. A procedure that is part of a relational database. The Data Access Builder can generate Java code that accesses stored procedures.

superclass. See *abstract class* and *base class*.

T

Transmission Control Protocol/Internet Protocol (TCP/IP). The basic programming foundation that carries computer messages around the globe through the Internet. The suite of protocols that defines the Internet. Originally designed for the UNIX operating system, TCP/IP software is now available for every major kind of computer operating system. To be truly on the Internet, your computer must have TCP/IP software.

tear-off property. A property that a developer has exposed to work with as though it were a stand-alone bean.

thread. A unit of execution within a process.

tool bar. The strip of icons along the top of the free-form surface. The tool bar contains tools to help an application developer construct composite beans.

transaction. In a CICS program, an event that queries or modifies a database that resides on a CICS server.

type. In VisualAge for Java, a generic term for a class or interface.

U

Unicode. A character coding system designed to support the interchange, processing, and display of the written texts of the diverse languages of the modern world. Unicode characters are normally encoded using 16-bit integral unsigned numbers.

Uniform Resource Locator (URL). A standard identifier for a resource on the World Wide Web, used by Web browsers to initiate a connection. The URL includes the communications protocol to use, the name of the server, and path information identifying the objects to be retrieved on the server. A URL looks like this:

<http://www.matisse.net/seminars.html>
or <telnet://well.sf.ca.us.br>
or <news:new.newusers.question.br>

user interface (UI). (1) The hardware, software, or both that enables a user to interact with a computer. (2) The term *user interface* normally refers to the visual presentation and its underlying software with which a user interacts.

V

variable. (1) A storage place within an object for a data feature. The data feature is an object, such as number or date, stored as an attribute of the containing object. (2) A bean that receives an identity at run time. A variable by itself contains no data or program logic; it must be connected such that it receives run-time identity from a bean elsewhere in the application.

versioned edition. An edition that has been versioned and can no longer be modified.

versioning. The act of making an open edition a versioned edition; that is, making the edition read-only.

view. (1) A visual bean, such as a window, push button, or entry field. (2) A visual representation that can display and change the underlying model objects of an application. Views are both the end result of developing an application and the basic unit of composition of user interfaces. Compare with *visual bean*. Contrast with *model*.

visual bean. In the Visual Composition Editor, a bean that is visible to the end user in the graphical user interface. Compare with *view*. Contrast with *nonvisual bean*.

visual programming tool. A tool that provides a means for specifying programs graphically. Application programmers write applications by manipulating graphical representations of components.

Visual Composition Editor. In VisualAge for Java, the tool where you can create graphical user interfaces from prefabricated beans and define relationships (connections) between both visual and nonvisual beans. The Visual Composition Editor is a page in the class browser.

W

Workbench. In VisualAge for Java, the main window from which you can manage the workspace, create and modify code, and open browsers and other tools.

workspace. The work area that contains all the code you are currently working on (that is, current editions). The workspace also contains the standard Java class libraries and other class libraries.

List of Abbreviations

ANSI	American National Standards Institute	IOR	interoperable object reference
API	application programming interface	ITSO	International Technical Support Organization
ATM	automated teller machine	JAR	Java archive
AWT	Abstract Windowing Toolkit	JDBC	Java Database Connectivity
CAE	Client Access Enabler	JDK	Java Developer's Kit
URL	uniform resource locator	JNI	Java Native Interface
CB	Component Broker	JVM	Java Virtual Machine
CICS	Customer Information Control System	LAN	local area network
CLI	call level interface	MOFW	managed object framework
COM	Component Object Model	MVS	Multiple Virtual Storage
CORBA	Common Object Request Broker Architecture	NLS	National Language Support
COS	CORBA object services	NT	new technology
DB2	DATABASE 2	ODBC	Open Database Connectivity
DBCS	double-byte character set	OMG	Object Management Group
DBMS	database management system	OMT	object modeling technique
DL/I	Data Language/I	OO	object-oriented
DLL	dynamic link library	OOA	object-oriented analysis
DNS	domain name server	OOD	object-oriented design
DRDA	Distributed Relational Database Architecture	ORB	Object Request Broker
ECD	edit-compile-debug	OS/2	Operating System/2
ECI	external call interface	OTS	object transaction service
FTP	File Transfer Protocol	PIN	personal identification number
GUI	graphical user interface	RAD	rapid application development
HTML	Hypertext Markup Language	RDBMS	relational database management system
HTTP	Hypertext Transfer Protocol	RMI	Remote Method Invocation
IBM	International Business Machines Corporation	SBCS	single-byte character set
IDE	integrated development environment	SDK	Software Developer's Kit
IDL	interface definition language	SQL	structured query language
IIOP	Internet inter-ORB protocol	TCP/IP	Transmission Control Protocol/Internet Protocol
IMS	Information Management System	TP	transaction processing
		UOW	unit of work
		URL	Uniform Resource Locator
		WWW	World Wide Web

Index

A

- access application 75
- account interface definition 337
- APPC 139, 364
- applet
 - ATM with JDBC 129
 - ATM with RMI 216
 - C++ sample 270
 - CBConnector sample 341, 346
 - CICS 246
 - code base 219
 - deployment 356
 - IString sample 282
 - JDBC 61
 - JDBC with Data Access Builder 80
 - RMI sample 157
 - viewer 51, 167, 211
- application
 - adapter framework 333
 - architectures 34
 - deployment 354, 355
 - design considerations 97
 - JDBC 89
 - layers 172
- array data 234, 242, 262, 294
- AS/400
 - feature 372
 - toolbox 372
- asynchronous processing 96
- ATM application
 - account information 105
 - business logic 109
 - C++ Access Builder 290
 - C++ implementation 291
 - C++ server integration 301
 - CICS Access Builder 229, 246
 - controller 195
 - Data Access Builder 99
 - data definition language 29
 - database 103
 - database design 21
 - design 20
 - design for distribution 172
 - distributed controller 216
 - entity-relationship diagram 21
 - flow 19, 129
 - introduction 17
 - layers 172

- main panel 130, 210
- object model 100, 174
- PIN validation 101
- referential integrity 27
- requirements 18
- RMI distribution 171, 213
- sample data 30
- sample run 212
- schema mapping 104
- service bean 192
- tables 24, 27
- test 211, 219
- transaction history 105
- user interface 115
- view layer 202
- VisualAge Generator 315
- wrapping C++ beans 299

- attribute 94
- authors xxvi
- autocommit 76

B

- BeanInfo 73, 154
- beans list 131, 158, 203
- bibliography 387
- BorderLayout manager 125
- business logic 110, 253
- business objects 182
 - data access classes 184
 - layer 174

C

- C++
 - server 255
 - templates 287
 - wrapper 265, 272, 279
- C++ Access Builder
 - advanced 272
 - ATM application 290
 - code generation 266
 - command line utility 266
 - compiler support 280
 - design considerations 276
 - environments 289
 - generated files 265, 273
 - header file modification 281
 - introduction 255
 - limitations 280
 - native example 267

-
- overview 12, 265
 - Version 1.0.1 379
 - CardLayout manager 81, 129
 - CB Toolkit 16, 330
 - CBConnector 16, 330
 - account example 338
 - creating objects 347
 - generating proxies 338
 - Java beans 342
 - Java client 337
 - VisualAge for Java 338
 - CICS 222
 - APPC 364
 - application coding techniques 241
 - client topologies 252
 - COBOL transaction 244
 - communications area 226
 - configuration 363
 - enterprise server 224
 - Gateway for Java 10, 221, 222, 250
 - installation 367
 - host access 221
 - topologies 252
 - installation 250, 362
 - Java application design 225
 - middleware 224
 - OS/2 362
 - program design 230
 - server simulation 249
 - unit of work 10, 225, 246, 248
 - CICS Access Builder
 - ATM application 229, 246
 - COBOL input 232
 - command line 239
 - concepts 223
 - data types 234
 - generated classes 241
 - introduction 221
 - invocation 237
 - overview 10, 226
 - restrictions 234
 - run-time class library 227
 - SmartGuide 238
 - Version 1.0.1 374
 - class
 - AccountDB 187
 - ATMApplet 130
 - AtmApplicationController 201
 - AtmDB 192
 - AtmDistributedController 215
 - BankAccount 112, 175
 - BuildCBC 342
 - CallableStatement 55
 - Card 110, 177, 292
 - CardDB 186
 - CardManager 291
 - CardPanel 115, 204
 - CheckingAccount 114, 176
 - Connection 49
 - CPPCard 299
 - Customer 177
 - CustomerDB 185
 - DriverManager 44
 - IMessageBox 297
 - IStrng 281
 - IVJCicsUOWInterface 227, 374
 - Naming 145
 - PinPanel 119, 205
 - PreparedStatement 54
 - ResultSet 49
 - ResultSetMetaData 49
 - SavingAccount 114
 - SavingsAccount 176
 - SelectAccountPanel 121, 206
 - Statement 49, 54
 - Transaction 178
 - TransactionDB 190
 - TransactionPanel 125, 208
 - UnicastRemoteObject 145
 - CLASSPATH 45, 89, 151, 166, 268, 284, 339, 354, 360, 367, 379
 - client/server
 - architecture 37
 - topology 252
 - COBOL
 - CICS Access Builder 232
 - CICS transaction 244
 - communications area 232
 - data types 234
 - code page 234, 248, 251, 378
 - COM.ibm.CORBA.iiop.ORB 343
 - COM.ibm.db2.jdbc.app.DB2Driver 45, 46
 - COM.ibm.db2.jdbc.net.DB2Driver 45, 48, 70, 108, 181, 193
 - COM.ibm.ivj.eab.data 74
 - COM.ibm.ivj.eab.j2cpp 279
 - COM.ibm.ivj.eab.rmi.client 152
 - COM.ibm.ivj.eab.rmi.server 152
 - COM.ibm.ivj.javabeans 358
 - COMMAREA
 - see communications area
 - communications area 226, 232, 363, 374
 - Component Broker 330
 - developing applications 335
 - introduction 16
 - programming model 331
 - currency service 328
 - controller 173, 195
 - distributed 216
 - CORBA 16, 317, 329

- services 325
- customized SQL statements 94

D

- data access 34
 - beans 89, 179
 - layer 38
- data access beans
 - business objects 184
- Data Access Builder
 - access application 75
 - advanced 92
 - ATM application 99
 - ATM application with RMI 180
 - BeanInfo 73
 - beans 89
 - concept 59
 - development process 63
 - export 92
 - generate 70
 - generated beans 72
 - identifier 68
 - import 92
 - key 68
 - overview 6
 - run stand-alone 92
 - sample application 79
 - schema mapping 65
 - share mappings 92
 - SmartGuide 66
 - starting 65
 - tree view 68
 - user-defined method 106
 - Version 1.0.1 373
 - window 65
- data definition language 29
- data identifier 68
- data logic 253
- data type
 - C++ mapping 277
 - JNI mapping 262
- database
 - ATM application 27
 - connection 84, 136
 - design 21
- datastore 72
- DB2 3
 - Client Application Enabler 42
 - JDBC daemon 46, 85, 360
 - JDBC drivers 45
 - prerequisites 360
 - sample database 47, 66

- Universal Database 55, 360
- db2java.zip 358, 360
- DB2JSTRT 46, 85, 360
- DBMS 36, 138
- debugger 2
- DFHCNV 374
- distributed processing 139
- DLL 257, 266, 268, 270
- driver
 - manager 40
 - see JDBC driver

E

- EAB
 - see enterprise access builders 3
- e-Business solution 314
- ECI 3, 306
- editor 2
- enterprise access builders 3
 - overview 4
 - run-time libraries 358
- entity 22
- entity-relationship diagram
 - ATM application 21
- event 110, 241
 - distributed 214
 - listener 110, 154
 - propagation 199
 - service 326
- exception 241, 279, 380
- export 166, 354, 356
- external call interface
 - see ECI
- externalization service 327

F

- fat client 36
- form 73
- framework 173
- free-form surface 192, 343

G

- garbage collector 263
- generic network protocol driver 43, 138
- GridBagLayout manager 115

H

header file 269
hosts file 165
HTML
 applet tag 50, 167

I

IDE 2
identity service 327
IDL 321, 323, 348
IIOP 321, 325
installation
 CICS 362
 CICS Gateway for Java 367
 redbook samples 368
 VisualAge for Java 359
integrated development environment
 see IDE
interchange file 92
interface definition language
 see IDL
ITSO publications 388
ivj2cpp 266, 269, 276, 379
ivjdata 92
ivjdcics 240
ivjeab.zip 358

J

jar 39, 358
Java Database Connectivity
 see JDBC
Java Native Method
 see JNI
Java Virtual Machine
 See JVM
JavaBeans 2
javah 258
JDBC 3
 applet 47, 50
 applet with VisualAge for Java 61
 application 48
 application structure 44
 concepts 33
 Data Access Builder 6, 60
 DB2 drivers 45, 70
 DB2 server daemon 46
 driver 39
 generic network protocol driver 43, 138

 ODBC bridge 41
 prerequisites 360
 sample program 46
 update program 51
 URL 46, 70
 vendor-specific driver 42
JDK 60, 256, 354
JNI 12, 255
 development process 261
 example 258
 exception handling 263
 overview 256
 programming 257
 type mapping 262
JVM 256, 257, 272, 354, 355

K

key
 foreign 25
 primary 25, 68

L

life cycle service 325
localhost 162, 181, 217
loopback 151

M

makefile 260
managed object framework 331
marshaling
 see serialization
method
 append 94
 asynchInvokeTxn 227
 bind 145
 createStatement 49
 execute 56
 executeQuery 49
 fill 94
 forName 45
 getMetaData 49
 getResultSet 56
 invokeTxn 227, 374
 loadLibrary 257
 native 257
 newInstance 45

- prepareStatement 55
- rebind 148
- setAsynchronous 96
- setSecurityManager 145
- simulateCICS 249
- toString 179
- Microsoft Visual C++ 280, 289
- middleware 38, 43, 304
- multicolumn list box 80
- MVS 221, 251, 306

N

- naming
 - server 145
 - service 325
- native method 257
- network protocol driver 138
- nonvisual beans 130
- normalization 23

O

- object
 - model 100, 174
 - services 333
- Object Builder 335
- Object Management Group 321
- Object Request Broker 139, 322
- ODBC 6, 34, 39, 41, 67
 - driver 361
 - prerequisites 361
- open class library 269
- Open Database Connectivity
 - see ODBC

P

- package 65, 237
- panel switching 131
- persistence
 - object service 328
- persistent
 - object 72
- Personal Communications 365
- personal identification number
 - see PIN 18
- PIN 18, 101, 197, 231
- port 46, 162, 360

- portability 354
- PowerServer API 15, 304, 305, 308
- prerequisites 360
- presentation logic 253
- promote features 119, 120, 124, 129
- proxy
 - bean 154
 - object 144
 - server 154
- proxy bean 154

Q

- query service 328

R

- rapid application development 307
- redbook samples 368
- referential integrity 24
- registry 143
- relational database access 60
- relationship 23
- remote
 - exception 147
- remote method invocation
 - see RMI
- Remote Object Instance Manager 152, 156, 164, 168
- remote procedure call 140
- remote reference layer 142
- reorder connections window 135
- reverse string 147, 269
 - applet 271
- RMI 3
 - account applet 161
 - architecture 141
 - client logic 149
 - compiler 143, 149
 - concepts 140
 - CORBA 329
 - design considerations 167
 - development process 143
 - distributed ATM application 219
 - distributed processing 139
 - execution environment 146, 155
 - limitations 169
 - native example 147
 - overview 8
 - problems and hints 165
 - proxy bean 154

- proxy object 144
- registry 143, 163
- Remote Object Instance Manager 152
- run application 151
- run outside VisualAge for Java 166
- security manager 145
- serialization 141
- server implementation 147
- special code 145
- stub and skeleton 142, 155
- technology 97
- tools 143
- trace 162, 164
- URL 149, 150
- VisualAge Generator 313
- RMI Access Builder
 - ATM application 171
 - development process 152
 - generated classes 154
 - overview 8
 - sample application 157
 - SmartGuide 152, 158, 215
 - VisualAge for Java 152
- rmic
 - see RMI compiler
- rmiregistry 151, 166
- rmiregst 151, 166
- rules
 - delete, insert, update 26

S

- schema mapping 65
- Scrapbook 201
- security
 - manager 145, 166
 - service 326
- serialization 141, 168, 213
- server proxy 154
- single-tier architecture 35
- skeleton 8, 142, 149
- SmartGuide
 - CB Toolkit 335
 - CICS Access Builder 226, 238
 - Data Access Builder 66, 102
 - Remote Program Call 372
 - RMI Access Builder 152, 158, 215
- SQL
 - command level interface 41
 - customized 94
 - data definition language 29
 - search predicate 95
 - statement 54

- statement validation 104
- stored procedure 96
- statement 54
 - callable 55
 - prepared 54
- stored procedure 55, 96
- stub 8, 142, 149
- synchronized 213
- system service layer 179

T

- TCP/IP 139
 - CICS 252
 - CICS configuration 363
 - hosts file 165
 - loopback 151
 - port 46, 162, 248, 360
 - RMI 140
- team programming 3
- tear-off property 192
- thin client 37
- three-tier architecture 37, 138, 223, 320, 360
- trace 162, 164, 165
- transaction service 327
- transport layer 142
- two-tier architecture 36, 138, 360

U

- URL
 - JDBC 46, 47, 70, 108, 193
 - RMI 149, 150

V

- variable 135, 192
- Version 1.0.1 371
 - migration 381
- version control 2
- Visual Composition Editor 84
 - concept 2
 - palette 396
- VisualAge for C++ 280, 289, 336
- VisualAge for Java
 - AS/400 feature 372
 - CBConnector 338
 - Enterprise

- connectivity 13
- introduction 1
- Entry 2
- import 94
- installation 359
- prerequisites 360
- products 2
- Professional 2
- repository 92
- Version 1.0.1 371
- VisualAge Generator
 - ATM application 315
 - client/server configuration 305
 - Common Services 310
 - CSO Java classes 307
 - e-Business solution 314
 - generated Java beans 308
 - introduction 15, 303
 - Java applet 310
 - Java application 309
 - Java client 307, 308
 - Java gateway 312, 313
 - Java support 304, 306
 - RMI 313
 - servers 304, 307
- VisualAge Smalltalk 306

W

- Web
 - browser 38, 39, 306, 356
 - server 38, 356
 - year 320
- Workbench 237, 354
- workload management 334

ITSO Redbook Evaluation

Application Development with VisualAge for Java Enterprise
SG24-5081-00

Your feedback is very important to help us maintain the quality of ITSO redbooks.
Please complete this questionnaire and return it using one of the following methods:

- ☐ Use the online evaluation form found at <http://www.redbooks.com>
- ☐ Fax this form to: USA International Access Code + 1 914 432 8264
- ☐ Send your comments in an Internet note to redbook@us.ibm.com

Please rate your overall satisfaction with this book using the scale:
(1 = very good, 2 = good, 3 = average, 4 = poor, 5 = very poor)

Overall Satisfaction _____

Please answer the following questions:

Was this redbook published in time for your needs? Yes___ No___

If no, please explain:

What other redbooks would you like to see published?

Comments/Suggestions: (THANK YOU FOR YOUR FEEDBACK!)

